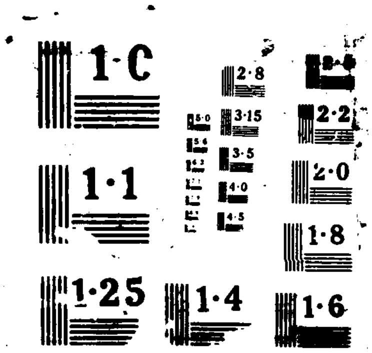
AD-A192 198 THE ISI (INFORMATION SCIENCES INSTITUTE) GRAPHER MANUAL (U) UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFORMATION SCIENCES INST G ROBINS FEB 88 1/2

UNCLASSIFIED ISI/TM-88-197 MDA903-81-C-0335 F/G 12/5 NL


1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·8
3·15
3·5
4·0
4·5

AD-A192 198

ISI Technical Manual
ISI/TM-88-197
February 1988

DTIC FILE COPY

4

University of Southern California

Gabriel Robins

The ISI Grapher Manual

DTIC ELECTE MAR 1 8 1988 S H D

INFORMATION SCIENCES INSTITUTE ISI

213-822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

88 3 16 090

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT: This document is approved for public release, distribution is unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): ISI/TM-88-197	5. MONITORING ORGANIZATION REPORT NUMBER(S): ---------------
6a. NAME OF PERFORMING ORGANIZATION: USC/Information Sciences Institute; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: ---------------
6c. ADDRESS (City, State, and ZIP Code): 4676 Admiralty Way, Marina del Rey, CA 90292	7b. ADDRESS (City, State, and ZIP Code): ---------------
8a. NAME OF FUNDING / SPONSORING ORGANIZATION: DARPA; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: MDA903-81-C-0335
8c. ADDRESS (City, State, and ZIP Code): DARPA, 1400 Wilson Blvd., Arlington, VA 22209	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
---------------	---------------	---------------	---------------

11. TITLE (Include Security Classification)

The ISI Grapher Manual [Unclassified]

12. PERSONAL AUTHOR(S) Robins, Gabriel

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Research Report	FROM ______ TO ______	1988, February	106

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	
09	02		artificial Intelligence tools, graph algorithms, graphs, intelligent systems, ISI Grapher, layout algorithms, user interfaces

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This document describes the implementation and usage of the ISI Grapher, a portable software package that allows graphs to be displayed pictorially. The salient features of the ISI Grapher are its speed, portability, extensibility, and versatility. The ISI Grapher currently runs on several different kinds of workstations (including Symbolics, TI Explorers, SUNS, and MacIntosh II), and is available commercially.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL: Sheila Coyazo, Victor Brown	22b. TELEPHONE (Include Area Code): 213-822-1511; 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

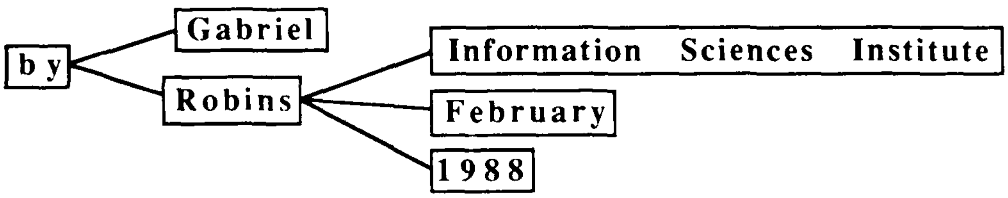
Unclassified

ISI Technical Manual
ISI/TM-88-197
February 1988

University of Southern California

Gabriel Robins

The ISI Grapher Manual



INFORMATION SCIENCES INSTITUTE

213/822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

This research is supported by the Defense Advanced Research Projects Agency under Contract No MDA903 81 C 0335 Views and conclusions contained in this report are the author's and should not be interpreted as representing the official opinion or policy of DARPA, the U S Government, or any person or agency connected with them.

Table of Contents

1. Introduction

This document describes the implementation and usage of the ISI Grapher, a portable tool for displaying graphs pictorially. The salient features of the ISI Grapher are its speed, portability, extensibility, and versatility. In the past few months, we received several hundreds of requests for the ISI Grapher from companies and universities worldwide, illustrating the substantial demand in both industry and in the research community for such a tool. The ISI Grapher currently runs on several different kinds of workstations (including Symbolics, TI Explorers, SUNs, and the MacIntosh II), and is available commercially.

In [Robins] it was demonstrated that the ability to interactively display and manipulate arbitrary directed graphs could greatly enhance end-user productivity, and a practical linear-time algorithm for laying out graphs was developed. For an introduction and an overview of the ISI Grapher, please refer to [Robins], as the current document assumes familiarity with the prior.

2. Users vs. Application Builders

Throughout this manual, the term *user* will be used to denote some person who is using the ISI Grapher (or some application which is built on top of the ISI Grapher, such as the NIKL Browser.) On the other hand, the term *application-builder* will be used to denote someone who is actually building an application using the ISI Grapher as a foundation. Users will be primarily interested in those sections of this manual which describe how to invoke the ISI Grapher (or systems which are built on top of it.) Application-builders, on the other hand, will want to pay special attention to those parts of this manual which describe how to modify and extend the ISI Grapher, describing how new applications may be easily supported using the ISI Grapher as a foundation.

3. Invoking the ISI Grapher

The ISI Grapher is invoked at the top-level by calling the function **graph-lattice**[1] with a list of roots/options and a "sons-function". This provides a means for the ISI Grapher to deduce the complete description of the graph by recursively calling the sons-function on the roots and their descendents[2].

Next, a reasonable graphical layout is computed for the graph, and is presented on the display. Various mouse sensitivity and functionality is automatically provided for, creating a

---

[1] throughout this document ISI Grapher keywords and function names will be bold-faced.

[2] other options and flags exist, and they will be described later; moreover, an extensive interface is provided below this high-level function, which applications may utilize when building on top of the ISI Grapher. This also will be described later in more detail later.

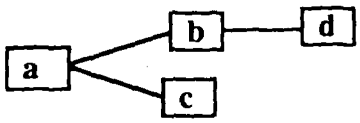
versatile and user-friendly browsing environment.

3.1. An Example

For example, if our graph is {(a,b),(a,c),(b,d)}, our root is {a}, and our sons-function is:

```
(defun sons (x)
      (cond ((eq x 'a) (list 'b 'c))
            ((eq x 'b) (list 'd))
            (t NIL)))
```

Note that the sons-function returns NIL if and only if the given node is a leaf in the graph (that is, the given node has no children.) Now, the call (**graph-lattice** 'a 'sons) would produce the picture of the graph:

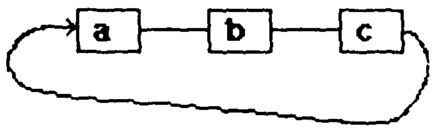


The arguments to **graph-lattice** are now summarized:

options - this may be a root object or an options list. If this argument is not a list, it is interpreted as the object corresponding to the root of the graph. If this argument is a list, it is interpreted as having the syntax and semantics described in the next section.

the-children-function - this argument is interpreted as the name of a (presumably existing) function which, when called with a single argument corresponding to a graph-object, returns the list of objects (of the same type) corresponding to the children of the argument node in the graph. That is, if 'sons is passed to **graph-lattice** as **the-children-function**, then anytime later, it is assumed that the call "(sons x)" returns a list "(y_1 y_2 ... y_n)" if and only if (x, y_i) is an edge in the graph, for all $1 \le i \le n$.

layout-flag - may be either **'tree** or **'lattice**: **'tree** means the graph will be displayed as a pure tree, regardless of its structure (in case there are cycles, they will be "broken" for displaying purposes by the introduction of "stub" nodes. For example, if this flag is **'tree** the graph {(a,b),(b,c),(c,a)} will really looks like:



will be actually displayed as:

where "**A**" represents the same graph node as does "a", so in a sense the graph node represented by "a" is displayed twice (with an obvious notation that this has occurred, such as the usage of a bolder font; this is automatically provided for by the ISI Grapher)., **'lattice** means that the given graph will be displayed with the cross-edges all displayed as they occur in the graph, with nodes properly displaced horizontally so that all edges are directed from left to right. If the graph contains any directed cycles, the **'tree** option will be automatically used[3]. In practice, **'tree** produces much tidier diagrams, while **'lattice** tends to depict more of the structure of graphs with many edges. If this flag is NIL or omitted, **'tree** is assumed. All directed edges are displayed with the direction going from left to right.

io-stream - this is the window/stream that will be used by the ISI Grapher to print messages to the user, and also to read character input typed by the user. If this argument is omitted, a reasonable default window will be used.

label - this string will become the resulting graph's description, to be used whenever a textual reference to that graph is required (such as in menu lines, etc.). If omitted, a standard default label will be used.

dont-track - this flag, if T, would cause the graph to be computed and displayed as it normally would be, except that after the corresponding graph window has been exposed, control would immediately transfer to the caller via a return; that is, **graph-lattice** would not track the mouse and accept grapher-related commands. This mode (i.e., with **dont-track** being T) of invocation of **graph-lattice** is very useful when the calling application would like to create a graph with its associated window, but *not* commence with an interactive browsing session; instead, the calling application would later "restart" the grapher (which will at that point is a trivial operation, since all the layout computations were already performed via the first call to **graph-lattice**.) If dont-track is NIL, interactive browsing mode will commence after the graph has been layed-out, and this is the default value for this optional argument.

the-parents-function - this is the analogue of **the-children-function**, and is used when transitive-closures are performed in the "up" (or reverse) direction on the graph; this hook is available just in case the "parents-function" is not exactly the inverse of the "children-function" for a particular graph. In summary, this argument should be the name of a function that when called with an object, should return the object's parents. If omitted, this function is computed from the semantics of **the-children-function** in the obvious way (that is, if X is a parent of Y if Y is a child of X).

3.2. Selective Pruning via the Options List

The first argument to **graph-lattice** may in fact be a command list with the following syntax, given here in BNF:

---

[3] so in fact the above example would be displayed as shown whether the layout flag was **tree** or **lattice**.

root-list ::= root | (root ...) | (command ...)
root ::= <LISP-object>
command ::= (keyword parameter ...)
keyword ::= **below** | **above** | **not** | **notbelow**
parameter ::= root | integer

where the keywords **below, above, not,** and **notbelow** may be abbreviated as **b**, **a**, **n**, and **nb**, respectively. These keywords have the following meanings:

below - Graph nodes below the given one(s).
above - *Graph nodes above the given one(s).*
notbelow - Do not graph nodes below the given one(s).
not - Do not include in the graph the given node(s).

Whenever an integer appears anywhere in place of a root, it causes the "search-depth" to be set to that integer. This determines how many levels down any (depth-first) search or transitive closure on the graph is computed. Originally the depth variable is set to "unbounded" and subsequent integers change the depth to the corresponding values. To make the depth unbounded again after it has been set to some value, include an arbitrary negative integer in the list at the point at which you would like this tc occur.

To compute the set of nodes to be graphed, a union is taken of all the **below** and **above** nodes, and a set difference is computed by subtracting all the **notbelow** and **not** nodes. These operations are done in the left-to-right order the options are specified, and hence may yield different graphs for different orders (as arbitrary sequences of the set operations of UNION and SUBTRACTION are *not associative*). This scheme provides a simple yet powerful method of selectively displaying exactly these parts of the graph which the user is interested in seeing.

We clarify the options-list syntax and its usage with some examples of well-formed root-lists:

((**below** *lispdata*))) - graph all nodes below the node *lispdata.*

lispdata - same as previous line.

((**b** *lispdata computerobject*) (**notbelow** *number file*) (**not** *process*)))
- graph all nodes below *lispdata* as well as those below *computerobject*, but do not include those nodes that are below *number* or *file*, and also not the node *process*.

((**a** *atom*)) - graph everything above (and including) *atom.*

((**below** 5 *thing*)) - graph everything below *thing* , but only to a depth of 5 levels.

((**below** 2 *x y* 13 *z* -1 *w*)) - graph everything below *x* and *y,* but only to a depth of 2 levels; also graph everything below *z* to a depth of 13 levels, and everything below *w* (without depth restriction).

A comment is in order here: if the first argument to **graph-lattice** is not a list, then it is taken to be the root of the graph. If it is a list, it is taken to be an options list with the above syntax/keywords. This means that if your graph has more than one root, it must be specified as

'((**below** a b c)) - note the double list.

4. The Main Command Menu

Once a graph has been layed-out and is displayed in a window, various commands are available from the main command menu. This menu is activated by clicking anywhere inside the currently active Grapher window. If the mouse cursor was pointing to a particular graph node during the mouse click, additional commands (tailored for and directed towards that particular node) shall become available on the main command menu. Appropriate documentation/explanation lines are available at the bottom of the display when the corresponding menu entry is highlighted. For the specific semantics of each such command, please refer to the corresponding function-description section in this document.

Clicking outside the currently-active grapher window has the following semantics: if the click occurred inside another (perhaps inactive/unexposed) Grapher window, then this window shall become exposed/activated and mouse tracking shall commence there with respect to the graph associated with that window. If the click occurred outside any Grapher window, the Grapher is suspended/exited until the user explicitly returns to it via an appropriate function call, or by pressing <terminal>-G or <function>-G, depending on what system the Grapher is running (the current convention is that <function>-G is used on Symbolics equipment, while <terminal>-G is used on TI hardware).

5. Performance and Efficiency

The time required by the ISI Grapher to lay-out a graph is linearly proportional to the size of the graph[4]. Moreover, the constant of proportionality in this linear relation is relatively small, yielding both a theoretical optimum, as well as practical efficiency[5]. In benchmark runs, speeds of up to 2,500 nodes per real-time minute have been achieved by the ISI Grapher when running on a Symbolics workstation (on relatively edge-sparse connected graphs, with the garbage collector turned off.)

It is further noted that there are numerous algorithms and heuristics to discretely lay-out graphs on the lattice-plane; however, the esthetic criterion that dictate what is a "nice" or "pleasing" layout vary greatly over users, and is very subjective. It can even be shown that under some simple esthetic assumptions, "optimal" layout becomes NP-hard (which in plain language

---

4 more formally, the asymptotic time (and space) complexity of the ISI Grapher for a graph $G=(V,E)$ is $O(|V| + |E|)$, where $|V|$ is the size of the node set, and $|E|$ is the size of the edge set.

5 note, however, that since the system must figure out the structure of the graph via multiple calls to the "sons-function", the efficiency of the system is ultimately reduced to to efficiency of the "sons-function." In the above discussion, we are assuming that the "sons-function" would return the set of sons of a given node in time proportional to the size of that set. It is guaranteed, however, that the ISI Grapher would not make more than $O(|V|)$ calls to the "sons-function," regardless of the possible existence of cycles in the graph.

means that no polynomial-time algorithms for such layouts are likely to exist) See, for example, [Supowit and Reingold].

The point of this discussion is to emphasize that the author does not advocate his layout scheme as the final word on such algorithms: he simply came into the belief (after considerable thought and experimentation with alternate layout schemes) that the scheme employed here is very close to being a relative-optimum on the curve of time efficiency vs. complexity (of specification) vs. esthetic appeal. In other words, the layout algorithm used here provides considerably more "beauty" (or "niceness") of layout per unit computation time, and is also quite simple to describe. For other layout schemes see, for example, [Wetherell and Shannon].

6. The Layout Algorithm

The layout algorithm employed by the ISI Grapher has several novel aspects. First, as previously mentioned, the asymptotic time and space performance of the layout algorithm is linear in the size of the graph being processed; this situation is clearly optimal. Second, the layout algorithm employed by the ISI Grapher exhibits an interesting symmetry: layout is performed independently in the X and Y directions. That is, first all the X coordinates (of the nodes in the layout) are computed, and then all the Y coordinates are computed **without** referring to the value of any of the X coordinates. This property implies a certain logical "orthogonality" in the treatment of the two planar dimensions, and is the source of the simplicity of the layout algorithm (the heart of the layout algorithm is only about two pages of code).

The Y coordinates of a node N is computed as follows: if N is a leaf node (that is, if N has no children in the graph) its Y coordinate is selected so that is it as close as possible to, but not overlapping any node previously layed out. If N has any children, their Y coordinates are computed first, and then N's Y coordinate is set to be the arithmetic average of the Y coordinates of N's children. Note that the second rule implies depth-first recursion, which is indeed how the algorithm is implemented. The Y-direction layout is sensitive to the heights of the objects being displayed. On the other hand, the Y-direction layout is completely oblivious to the X-coordinate values.

Similarly, the X coordinates of a node N is computed as follows: if N is a root node (that is, if N has no parents in the graph), its X coordinate is set to zero. If N has any parents, their X coordinates are computed first, and then N's X coordinate is set to be some fixed amount larger than the maximum of the X coordinates of N's parents. Again, note that this implies depth-first recursion. The X-direction layout is sensitive to the lengths of the objects being displayed, and is completely oblivious to the Y-coordinate values.

For the sake of completeness, we specify the X and Y layout algorithms more formally. The layout algorithm for the Y coordinates is specified as follows:

```
For N in Nodes do Y[N] := 0;
Last-y := 0;
For N in Roots(G) do Layout-Y(N);

Procedure Layout-Y(N);
```

```
begin
if Y[N] = 0 then                        /* N was not yet layed-out */
        If N has any unlayed-out children then
                begin                   /* layout the children first. */
                for C in Children(N) do Layout-Y(C);
                Y[N] := average-Y(Children(N));
                end
        else    begin                   /* layout a leaf. */
                Y[N] := Last-y + Height(N);
                Last-Y := Y[N];
                end;
end;  /* of procedure Layout-Y */
```

The layout algorithm for the X coordinates is specified as follows:

```
For N in Nodes do X[N] := 0;
For N in Leaves(G) do Layout-X(N);

Procedure Layout-X(N);
begin
if X[N] = 0 then                        /* N was not yet layed-out. */
        If N has parents then
                begin                   /* layout the parents first. */
                for C in Parents(N) do Layout-X(C);
                X[N] := Max{X[i] + Width(i) | i in Parents(N)} + constant;
                end
end;  /* of procedure Layout-X */
```

From the recursive layout scheme specified above, it should be clear that each node gets processed only once during the two independent passes (one for each of the two coordinate axes.)

7. The Data Structures

The ISI Grapher maintains various data structures for each graph that it processed. In particular, each node and edge of the graph is represented as an instance of a LISP record structure. Various useful information is maintained in each record and thus may be directly extracted whenever a pointer to such a record is available. The fields contained in each node-record include the object the node represents, the print-name of the object and its dimensions, the associated font, the location on the screen of the node, as well as the children and parents of this node. The fields contained in each edge-record include the node from which the given edge emanates and the node upon which the given edge terminates (edges are directed).

Another kind of record, called a window-record, is maintained for each grapher window, and includes a list of records corresponding to the roots of the graph, the name of the children-function, a list of node- and edge-records associated with the given graph, a list of available fonts, various size parameters for the graph and its window, and a list corresponding to the subset of the

nodes and edges that are currently visible (or partially visible) in the graph window. Essentially, each window-record contains a copy of each global variable associated with a single graph, its layout, and its window.

Several hash tables are maintained by the grapher. The most important of these is the name-to-node table which maps objects to the node-record associated with them. Application-builders need not refer to this table directly, however, as several utility functions for mapping/translating various objects to other objects are provided for and are described elsewhere in this document.

Application builders are reminded that any code that they write should leave these various data structures in a consistent state. It is therefore advisable that application-builders use the built-in functions, whenever possible, for such manipulations, as opposed to writing their own. Of course, in some cases where the needs of an application-builder are very specialized, getting "one's hands dirty" with the internals of the ISI Grapher may be quite unavoidable.

8. The Control Structure

Once a graph has been layed-out and is displayed in a window, various commands are available from the main command menu. In addition, many more functions are available for the application-builder's use. The ISI Grapher is initially called/entered with the high-level function **graph-lattice** (or with some function which calls it); this is described in more detail elsewhere in this manual. Subsequent entries into the Grapher may be achieved very quickly via pressing <terminal>-G or <function>-G.

If a mouse-click occurred inside another (perhaps inactive/unexposed) Grapher window, then this window shall become expose/activated and mouse tracking shall commence there. If a click occurred outside any Grapher window, the Grapher is suspended/exited. The global variable **tracking-mouse** is set to non-NIL if and only if the grapher is currently tracking the mouse. An alternate way of exiting the grapher (from within an application running concurrently with the grapher) is to set the global variable **tracking-mouse** to NIL. Yet another way of exiting the Grapher entails selecting the "suspend" command from the main command menu.

When the mouse points in an active Grapher window to some node, that node becomes highlighted and various additional commands from the main command menu become available and operate with respect to that node; for example, if a node is selected (highlighted) and the command "delete-node" is issued by selecting the corresponding menu item, that node will be removed from the graph and the window will be redrawn (if any nodes have become orphans/parentless as a result of this operation, they too will be so removed from the graph, recursively).

9. Portability and Code Organization

In trying to keep the ISI Grapher as portable as possible, the code is divided into two main modules. The first and largest module consists of pure Common LISP code; this code is responsible for all the layout, control, and data-structure manipulation algorithms. The second module is substantially smaller, and consists of numerous low-level primitive calls which are quite likely to be implementation-dependent. The intent here is that when the Grapher is to be ported to another (Common LISP) environment, **only** the second module should require modification. In

order to further minimize porting efforts, the calls from code in the first module to functions in the second module were designed to be as generic as possible.

In summary, if a new environment has a window-system which supports a reasonable set of window and graphics primitives (such as open-window, draw-line, print-string, etc.), then porting the ISI Grapher to this new environment or machine should require a minimal coding effort, probably all of which would be confined to the second section of the ISI Grapher code.

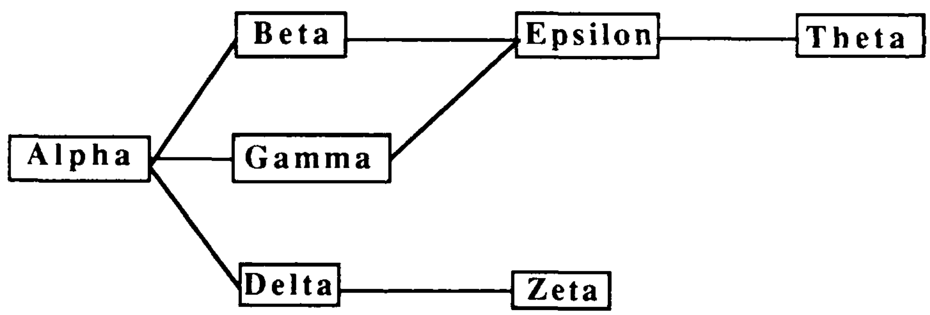
10. Some Applications

As examples of how easily other applications may be built on top of the ISI Grapher, several such applications have already been built and are loaded along with the ISI Grapher. We now describe these applications:

The List Grapher - This application displays the natural correspondence between lists and trees. For example, the call

```
(graph-list   '(alpha (beta (epsilon theta))
                      (gamma epsilon)
                      (delta zeta))
              'lattice)
```

would produce the following picture:



This provides an easy means of quickly obtaining large or complex graphs.

<u>The Flavor Grapher</u> - This application displays the interdependencies between flavors, where nodes are flavor names, and edges mean "depends on." This type of a diagram could be quite useful in system development. For example, the call:

```
(graph-flavor 'tv:window 'lattice)
```

would graph (as a lattice) all the flavors that depend on the tv:window flavor, while the call

```
(graph-flavor 'si:vanilla-flavor)
```

would graph the entire flavor hierarchy (which is likely to be quite large).

<u>The Package grapher</u> - This application produces a picture of the package interdependencies between a package and all packages that use it. This picture could be illuminating to anyone who have had to struggle with the awkward semantics of package inheritance. An example of a call is:

```
(graph-package "global")
```

<u>The Divisors Grapher</u> - This application displays the divisibility graph of a given integer; that is, all the divisors of an integer are represented nodes, where an edge between two nodes means "is divisible by." This is also a quick method to produce large graphs. Examples follow:

```
(graph-divisors 360)

(graph-divisors 360 'lattice)

(graph-divisors 5040)
```

To illustrate how easily such tools may be built on top of the ISI Grapher, we note in passing that coding and testing all 3 tools (the flavor grapher, the package grapher, and the divisibility grapher) took about half an hour of work.

<u>The NIKL Browser</u> - This application is a browsing tool for NIKL networks. The function **nkbc** graphs a taxonomy below a given concept list with various options; the call (**nkbc**) graphs the entire NIKL concept taxonomy. (see other examples in the description of **graph-lattice**). Similar syntax holds for graphing role taxonomies using **nkbr**. The function **nkb** graphs either roles or concepts, and it tries to figure out which you meant from the names. Note that if you are in the wrong package, it may be necessary to specify concept names with the "nikl" package prefix; for example, "nikl:lispdata" (or "nikl::lispdata" on TI workstations), etc.

11. Application-building

11.1. Adding to the main command menu

To add a new user-defined function 'foo to the command menu, make the call:

```
(add-to-command-menu
    "Do my-op"
    'MyOp
    "MyOp does the following: ...")
```

The first argument here is the menu-line, the second argument is the function to be called with the highlighted graph object (or NIL if nothing is highlighted) as well as the window in which the clicking occurred, and the third argument is the documentation line for this menu item. This documentation line will appear on the screen when the mouse/cursor is pointing at that menu item.

For example, suppose the application-builder would like to add to the main command menu a

functionality that will accept a graph node and pretty-print the associated object, using some special user-defined pretty-printer function, called "user-pp." The application-builder may then evaluate a form resembling the following:

```
(add-to-command-menu    "Do a special PP of this node"
    'user-pp
    "Does a nifty PP of this graph node.")
```

and the function "user-pp" may be defined as follows:

```
(defun user-pp (node window)
    ; Window is ignored, but it must be accommodated for in the
    ; formal argument list anyway.
    (setq the-object (node-name node))      ; get the structure.
    (special-user-pp  the-object)           ; pp the structure.
    . . .
    )
```

To reset the main command menu to its original default state, simply execute the following function: **(initialize-command-menu).** This will permanently get rid of all application-builder-defined commands in the main command menu. Note that an application-builder-defined command menu may display a secondary menu once it executes, producing a logical menu hierarchy. This strategy can be used in order to prevent the main command menu from getting too cluttered with numerous low-level commands.

11.2. Overriding Default Operations

Several basic Grapher operations may be controlled via the specification of alternate functions for performing these tasks. These operations include the drawing of nodes and edges, the selections of fonts, the determination of print-names, pretty-printing, and highlighting operations. Standard definitions are already provided for these operations and are used by default if the application-builder does not override them by specifying his own functions for performing these tasks.

For example, the default method of highlighting a graph node when the cursor points to it on the screen is to invert a solid rectangle of bits over the node. Suppose that the user is not satisfied with this mode of highlighting and would like to have thin boxes drawn around highlighted nodes instead. He may write a highlighting function that does exactly that, and tell the Grapher to use that function whenever a node needs to be highlighted. The details and semantics of this process will be further explained shortly.

As another example, suppose the user is not happy with the way nodes are displayed on the screen; ordinarily nodes are displayed on the screen by printing their ASCII print-names at their corresponding screen location, but the user would prefer that some specialized icon be displayed instead. The user may then specify his icon-displaying function as the normal node-painting function and from then on, whenever a node needs to be displayed on the screen, that function will be called upon (along with arguments corresponding to the node, its screen location, and the relevant window) thus achieving the desired effect.

In particular, the following basic Grapher operations may be overridden by the user:

- Deciding which font should be used to display an object's print-name. Different fonts may thus be used to distinguish various types of objects.

- Determining the dimensions (width and height) of an object. This information is used by the other Grapher functions, such as the layout algorithm (as placement of objects is sensitive to their sizes) and highlighting operations (as the size of the highlight-box depends on the size of the object being highlighted.)

- Determining the ASCII print-name of an object.

- Highlighting and unhighlighting an object. This operation is most often performed when the mouse points to a given object.

- Describing or explaining an object. This is the function that gets executed when the corresponding explain (or pp) command is selected from the main menu.

For each one of the categories above, the Grapher keeps a *function precedence list*, consisting of a primary function, a secondary function, a tertiary function, and so on, for as many functions as are currently available to perform the task associated with that particular category. Whenever a new function is introduced to perform a certain task, it is made the primary function for that category, while the previous primary function is made the secondary function, and so on (i.e., each function is "demoted" one "notch" in precedence). In addition, each category also has associated with it a default function, which is initially the only function associated with that category (that is, initially there is no primary function, nor secondary function, etc.) The default function for a particular category has the least precedence relative to any other functions in that category.

When a certain task needs to be performed during the normal operation of the Grapher, the corresponding primary function is called with a graph node object[6] and a window. It is then up to the called function to perform the given task and return non-NIL if it indeed performed the said task, or NIL it is did not (or could not or chose not to) perform the said task. In the former case the Grapher merrily goes about its business, while in the latter case, the secondary function is similarly called, with this process repeating until some function successfully performed the given task (this event being signaled by the return of non-NIL by that function.), or until all the available functions have been exhausted and the task has not yet been performed. In the latter case the default function is called, the default function being guaranteed to perform the associated task successfully.

This mechanism gives the user great flexibility in displaying and highlighting graph objects. These operations may depend heavily on the type and size of the object being displayed or highlighted, and so different functions may be used to handle each type of object. It should be noted that this discussion implies the ability to mix various types of objects in the same graph (each having unique size, appearance, and highlighting characteristics) with relative ease and uniformity. In summary, this scheme is reminiscent of a primitive *flavor* (or object-oriented) mechanism, where "inheritance" has a non-standard semantics.

---

[6] this is a complex structure from which a lot of other information may be determined; the details are described in the section entitled "Node and Edge Objects."

Each one of the outlined categories has a family of functions associated with it. In particular, for each operation-type <op> associated with one of the above categories, the following functions are defined:

- **add-<op>-function** - this function, which takes a function name as an (optional) argument, adds the specified function as the primary function for the corresponding category, after "demoting" each such previously defined function one "notch" in precedence, as described above. If the argument to **add-<op>-function** is NIL, no action is taken, but in either case the current list of functions associated with that category is returned, sorted by descending precedence, with the default-function taken to have the least precedence relative to the other functions for that category.

- **init-<op>-function-list** - this argumentless function deletes all the functions associated with the particular category, except the default-function. This provides a means to (re-)initialize the function list associated with a particular category. Selective deletion of particular functions from the precedence list for a particular category may be accomplished via initialization (that is, a call to **init-<op>-function-list)** and the subsequent additions of the functions that are to remain.

- **default-<op>-function** - this function constitutes the default function for the corresponding operation. It is usually called with two arguments, a graph node object and a window. In some categories, the returned result is independent of the window (the second argument), but in other categories the second argument is essential; for the sake of *uniformity (and future expansion), however, we pass* both arguments to each function in this family. In addition, some members of this family of functions are called with additional arguments; the specifics will be described later.

In the above discussion, <op> may be one of {**font**, **pname**, **pname-length**, **pname-height**, **highlight-node**, **unhighlight-node**, **describe**, **node-paint**, **edge-paint**}. In other words, the above family of functions is parametrized by this set of keywords.

Keep in mind that the arguments "node" and "edge" are LISP structures from which various other information may be extracted, such as coordinates, fonts, pnames, etc. In particular, the containing-window of an object is also stored in the associated structure; this means that the second argument, being "window" to to some of the functions is superfluous. We therefore use the following convention: if the window argument is specified, it overrides the window specified in the given (node or edge) structure. This enables performing some operations to arbitrary windows, as opposed to just the windows containing the actual objects. This facility is quite useful in scrolling, for example, when parts of the graph need to be displayed in a special scroll-window.

When the application-builder supplies his own function for a particular category, that function must externally mimic the semantics for that category. In particular, the application-builder's function must have the same number (and type) of arguments (and returned value) as the default function for that category. For example, if the application-builder defines a new font-function, it must accept two arguments, a node object and a window, and return a font. How the returned font is selected (or whether it depends on the input at all), is a decision left entirely to the application-builder.

The application-builder must exercise some care, however, in designing his replacement functions; for example, in most applications, the unhighlight-function should "undo" what the highlighting-function does, etc. The Grapher cannot determine whether the application-builder has provided a consistent (or a useful) set of functions. The moral of this discussion is that "with great freedom comes great responsibility."

In summary, many of the basic Grapher operations are parametrized by a set of default methods. This set may be extended by the application-builder in order to make the ISI Grapher behave in ways not provided for by the author. Any operations left unspecified by the application-builder will default to some reasonable pre-defined method. This scheme makes for a very flexible and extendible system.

For the sake of completeness and clarity, we now list the entire family of functions described by the above scheme, as well as their arguments, side effects, and descriptions:

11.2.1. add-describe-function

Argument: describe-function (optional)

Returns: The current list of describe-functions, sorted by descending precedence, with the default describe-function having the least precedence.

Side effects: If no argument is supplied, there are no side effects; otherwise, describe-function, which is assumed to be the name of an existing (user-supplied) function, is added as the primary describe-function. The previously primary describe-function now becomes the secondary describe-function, and so on (i.e., each previously defined describe-function is demoted one "notch" in precedence.)

Description: Whenever the Grapher needs to produce a description of the object associated with a particular node, the describe-function(s) will be called to produce this description, using the precedence semantics described previously. The function describe-function is assumed to take two arguments, a graph node object and a window, and print a description (corresponding to the object associated with the given node). Note that the description process is a dynamic one, so the describe-function is called only when it is needed.

11.2.2. add-edge-paint-function

Argument: edge-paint-function (optional)

Returns: The current list of edge-paint-functions, sorted by descending precedence, with the default edge-paint-function having the least precedence.

Side effects: If no argument is supplied, there are no side effects; otherwise, edge-paint-function, which is assumed to be the name of an existing (user-supplied)

function, is added as the primary edge-paint-function. The previously primary edge-paint-function now becomes the secondary edge-paint-function, and so on (i.e., each previously defined edge-paint-function is demoted one "notch" in precedence.)

Description: Whenever the Grapher needs to paint an edge on any window of the display, the edge-paint-function(s) will be called to perform this task, using the precedence semantics described previously. The function edge-paint-function is assumed to accept four arguments: an edge, a window, an x-offset, and a y-offset. It is further expected that this function would return non-NIL if the edge-painting operation was performed successfully, or else return NIL if it did not perform the edge-painting operation for some reason. This returned value will be used to determine if another edge-paint function needs to be called to complete the operation. Edge-painting is usually performed when a Grapher window is (re)displayed and also during scrolling operations.

11.2.3. add-font-function

Argument: font-function (optional)

Returns: The current list of font-functions, sorted by descending precedence, with the default font-function having the least precedence.

Side effects: If no argument is supplied, there are no side effects; otherwise, font-function, which is assumed to be the name of an existing (user-supplied) function, is added as the primary font-function. The previously primary font-function now becomes the secondary font-function, and so on (i.e., each previously defined font-function is demoted one "notch" in precedence.)

Description: Whenever the Grapher needs to determine what font to assign a particular node, the font-function(s) will be called to determine this font, using the precedence semantics described previously. The function font-function is assumed to take two arguments, a node and a window, and return a font that will be used in the future, whenever the print-name of the given node has to be displayed. Note that the font selection process is a static one; that is, all the node fonts are selected and recorded only once, *before* the graph is ever displayed for the first time. Thus, changing the font-function will *not* affect the fonts associated with the nodes of an already existing graph.

11.2.4. add-highlight-node-function

Argument: highlight-node-function (optional)

Returns: The current list of highlight-node-functions, sorted by descending precedence, with the default highlight-node-function having the least precedence.

Side effects: If no argument is supplied, there are no side-effects; otherwise, highlight-

node-function, which is assumed to be the name of an existing (user-supplied) function, is added as the primary highlight-node-function. The previously primary highlight-node-function now becomes the secondary highlight-node-function, and so on (i.e., each previously defined highlight-node-function is demoted one "notch" in precedence.)

Description: Whenever the Grapher needs to highlight a particular node on the display, the highlight-node-function(s) will be called to perform this task, using the precedence semantics described previously. The function highlight-node-function is assumed to accept two arguments, a node and a window; it is further expected that this function would return non-NIL if the highlighting operation was performed successfully, or else return NIL if it did not perform the highlighting operation for some reason. This returned value will be used to determine if another highlighting function needs to be called to complete the operation. Highlighting of a node is usually performed when the mouse points to that node on the screen.

11.2.5. add-node-paint-function

Argument: node-paint-function (optional)

Returns: The current list of node-paint-functions, sorted by descending precedence, with the default node-paint-function having the least precedence.

Side effects: If no argument is supplied, there are no side-effects; otherwise, node-paint-function, which is assumed to be the name of an existing (user-supplied) function, is added as the primary node-paint-function. The previously primary node-paint-function now becomes the secondary node-paint-function, and so on (i.e., each previously defined node-paint-function is demoted one "notch" in precedence.)

Description: Whenever the Grapher needs to paint a node on any window of the display, the node-paint-function(s) will be called to perform this task, using the precedence semantics described previously. The function node-paint-function is assumed to accept four arguments, a node, a window, an x-offset, and a y-offset; it is further expected that this function would return non-NIL if the node-painting operation was performed successfully, or else return NIL if it did not perform the node-painting operation for some reason. This returned value will be used to determine if another node-paint function needs to be called to complete the operation. Node-painting is usually performed when a Grapher window is (re)displayed and also during scrolling operations.

11.2.6. add-pname-function

Argument: pname-function (optional)

Returns: The current list of pname-functions, sorted by descending precedence, with

the default pname-function having the least precedence.

Side effects: If no argument is supplied, there are no side-effects; otherwise, pname-function, which is assumed to be the name of an existing (user-supplied) function, is added as the primary pname-function. The previously primary pname-function now becomes the secondary pname-function, and so on (i.e., each previously defined pname-function is demoted one "notch" in precedence.)

Description: Whenever the Grapher needs to determine what print-name to associate with a particular node (for displaying operations), the pname-function(s) will be called to determine this print-name, using the precedence semantics described previously. The function pname-function is assumed to take two arguments, a graph node object and a window, and return a string (corresponding to the print-name of that object) which will be used during all future layout and display operations. Note that the print-name determination process is a static one; that is, all the node print-names are determined and recorded only once, *before* the graph is ever displayed for the first time. Thus, changing the pname-function will *not* affect the print-names associated with the nodes of an already existing graph.

11.2.7. add-pname-height-function

Argument: pname-height-function (optional)

Returns: The current list of pname-height-functions, sorted by descending precedence, with the default pname-height-function having the least precedence.

Side effects: If no argument is supplied, there are no side-effects; otherwise, pname-height-function, which is assumed to be the name of an existing (user-supplied) function, is added as the primary pname-height-function. The previously primary pname-height-function now becomes the secondary pname-height-function, and so on (i.e., each previously defined pname-height-function is demoted one "notch" in precedence.)

Description: Whenever the Grapher needs to determine the height of the print-name associated with a particular node (for layout, displaying, and highlighting operations), the pname-height-function(s) will be called to determine this height, using the precedence semantics described previously. The function pname-height-function is assumed to accept two arguments, a node and a window, and return an integer corresponding to the height of the print-name (in display-units, which are normally pixels.) This value will be used in the future, whenever the given node has to be layed-out, displayed, or highlighted. Note that the print-name-height determination process is a static one; that is, all the node print-name heights are determined and recorded only once, *before* the graph is ever displayed for the first time. Thus, changing the pname-height-function will *not* affect the print-name heights associated with the nodes of an already existing graph.

11.2.8. add-pname-length-function

Argument: pname-length-function (optional)

Returns: The current list of pname-length-functions, sorted by descending precedence, with the default pname-length-function having the least precedence.

Side effects: If no argument is supplied, there are no side-effects; otherwise, pname-length-function, which is assumed to be the name of an existing (user-supplied) function, is added as the primary pname-length-function. The previously primary pname-length-function now becomes the secondary pname-length-function, and so on (i.e., each previously defined pname-length-function is demoted one "notch" in precedence.)

Description: Whenever the Grapher needs to determine the length of the print-name associated with a particular node (for layout, displaying, and highlighting operations), the pname-length-function(s) will be called to determine this length, using the precedence semantics described previously. The function pname-length-function is assumed to accept two arguments, a node and a window, and return an integer corresponding to the length of the print-name (in display-units, which are normally pixels.) This value will be used in the future, whenever the given node has to be layed-out, displayed, or highlighted. Note that the print-name-length determination process is a static one; that is, all the node print-name lengths are determined and recorded only once, *before* the graph is ever displayed for the first time. Thus, changing the pname-length-function will *not* affect the print-name lengths associated with the nodes of an already existing graph.

11.2.9. add-unhighlight-node-function

Argument: unhighlight-node-function (optional)

Returns: The current list of unhighlight-node-functions, sorted by descending precedence, with the default unhighlight-node-function having the least precedence.

Side effects: If no argument is supplied, there are no side-effects; otherwise, unhighlight-node-function, which is assumed to be the name of an existing (user-supplied) function, is added as the primary unhighlight-node-function. The previously primary unhighlight-node-function now becomes the secondary unhighlight-node-function, and so on (i.e., each previously defined unhighlight-node-function is demoted one "notch" in precedence.)

Description: Whenever the Grapher needs to unhighlight a previously highlighted node on the display, the unhighlight-node-function(s) will be called to perform this task, using the precedence semantics described previously. The function

unhighlight-node-function is assumed to accept two arguments: a node and a window. It is further expected that this function would return non-NIL if the unhighlighting operation was performed successfully, or else return NIL if it did not perform the unhighlighting operation for some reason. This returned value will be used to determine if another unhighlighting function needs to be called to complete the operation. Unhighlighting of a node is usually performed when the mouse no longer points to a previously highlighted node on the screen.

11.2.10. default-describe-function

Argument 1: a node

Argument 2: a window (optional)

Returns: t

Side effects: none

Description: This is the default describe-function, having the least precedence relative to any other describe-functions. It is used by the grapher to produce and display a standard description of the object associated with the given node. It is called by the Grapher as a last resort, when none of the other describe-functions have produced a description (or when no other describe-functions exist.) In most cases an application-builder would want to provide his own describe function, which possesses knowledge about the objects associated with the nodes of the graph.

11.2.11. default-edge-paint-function

Argument 1: an edge

Argument 2: a window (optional)

Argument 3: x-offset, an integer(optional)

Argument 4: y-offset, an integer (optional)

Returns: t

Side effects: The specified edge is painted in the given window, using the given offsets.

Description: This is the default edge-paint-function, having the least precedence relative to any other node-paint-functions. It is used by the grapher to display an edge at a given window and is guaranteed to actually perform this operation (and return non-NIL.) It is called by the Grapher as a last resort, when none of the other edge-paint-functions have returned non-NIL (or when no other edge-

paint-functions exist.) The default method of displaying an edge is to draw a line between the two nodes which define the edge. The x- and y- offsets (if given) are added to the actual coordinates of the edge being drawn for the purpose of this operation; the actual coordinates of the edge remain unaffected. This gives extra flexibility in choosing where edges will be drawn.

11.2.12. default-font-function

Argument 1: a node

Argument 2: a window (optional)

Returns: a font

Side effects: none

Description: This is the default font-function, having the least precedence relative to any other font-functions. It is used by the grapher to choose a font for the print-name of a particular node, and is guaranteed to return a font. It is called by the Grapher as a last resort, when none of the other font-functions have returned a font (or when no other font-functions exist.) The font this function returns corresponds to some standard plain-looking font.

11.2.13. default-highlight-node-function

Argument 1: a node

Argument 2: a window (optional)

Returns: t

Side effects: The specified node is highlighted in the given window.

Description: This is the default highlight-node-function, having the least precedence relative to any other highlight-node-functions. It is used by the grapher to highlight a particular node, and is guaranteed to actually perform this operation (and return non-NIL.) It is called by the Grapher as a last resort, when none of the other highlight-node-functions have returned non-NIL (or when no other highlight-node-functions exist.) The default manner of highlighting is to xor a rectangle of bits (with width and length determined by the pname-length-function and pname-height-function, respectively) onto the screen at the location corresponding to the given node. This is indeed the highlighting scheme utilized by this function, and it has the interesting property that to un-highlight a node, one needs only to repeat this operation (that is, this operation is equivalent its own inverse.)

11.2.14. default-node-paint-function

Argument 1: a node

Argument 2: a window (optional)

Argument 3: x-offset, an integer (optional)

Argument 4: y-offset, an integer (optional)

Returns: t

Side effects: The specified node is painted in the given window, using the given offsets.

Description: This is the default node-paint-function, having the least precedence relative to any other node-paint-functions. It is used by the grapher to display a node at a given window and is guaranteed to actually perform this operation (and return non-NIL.) It is called by the Grapher as a last resort, when none of the other node-paint-functions have returned non-NIL (or when no other node-paint-functions exist.) The default method of displaying a node is to print its print-name (using the proper font) at the corresponding screen coordinates. The x- and y- offsets (if given) are added to the actual coordinates of the node being drawn for the purpose of this operation; the actual coordinates of the node remain unaffected. This gives extra flexibility in choosing where nodes will be drawn.

11.2.15. default-pname-function

Argument 1: a node

Argument 2: a window (optional)

Returns: a string (corresponding to a print-name)

Side effects: none

Description: This is the default pname-function, having the least precedence relative to any other pname-functions. It is used by the grapher to choose a print-name for a particular node, and is guaranteed to return a print-name. It is called by the Grapher as a last resort, when none of the other pname-functions have returned a print-name (or when no other pname-functions exist.) The print-name it returns is a string which represent the given node, and is similar to what the LISP (format NIL "~A" X) would return, where X is the object whose print-name we are seeking.

11.2.16. default-pname-height-function

Argument 1: a node

Argument 2: a window (optional)

Returns: an integer (corresponding to the print-name height)

Side effects: none

Description: This is the default pname-height-function, having the least precedence relative to any other pname-height-functions. It is used by the grapher to determine the print-name-height for a particular node, and is guaranteed to return an integer. It is called by the Grapher as a last resort, when none of the other pname-height-functions have returned a print-name height (or when no other pname-height-functions exist.) The print-name-height it returns is an integer which corresponds to the height (in display-units, which are normally pixels) of the print-name of the given graph node object.

11.2.17. default-pname-length-function

Argument 1: a node

Argument 2: a window (optional)

Returns: an integer (corresponding to the print-name length)

Side effects: none

Description: This is the default pname-length-function, having the least precedence relative to any other pname-length-functions. It is used by the grapher to determine the print-name-length for a particular node, and is guaranteed to return an integer. It is called by the Grapher as a last resort, when none of the other pname-length-functions have returned a print-name length (or when no other pname-length-functions exist.) The print-name-length it returns is an integer which corresponds to the length (in display-units, which are normally pixels) of the print-name of the given graph node object.

11.2.18. default-unhighlight-node-function

Argument 1: a node

Argument 2: a window (optional)

Returns: t

Side effects: The specified node is unhighlighted in the given window.

Description: This is the default unhighlight-node-function, having the least precedence relative to any other highlight-node-functions. It is used by the grapher to unhighlight a previously highlighted node, and is guaranteed to actually perform this operation (and return non-NIL.) It is called by the Grapher as a last resort, when none of the other unhighlight-node-functions have returned non-NIL (or when no other unhighlight-node-functions exist.) The unhighlighting operation should *undo* what the highlighting operation does. The default manner of unhighlighting is therefore to xor a rectangle of bits (with width and length determined by the pname-length-function and pname-height-function, respectively) onto the screen at the location corresponding to the given node. This is indeed the unhighlighting scheme utilized by this function, and it has the interesting property that the highlighting and unhighlighting operations have the same semantics.

11.2.19. init-describe-function-list

Arguments: none

Returns: nothing of significance

Side effects: All the describe-functions are deleted (i.e., forgotten), except for the default-function.

Description: This function provides a means to (re-)initialize (or clear) the describe-function precedence list. Deleting only particular describe-functions from the precedence list entails initializing the list and then adding the ones that should not have been deleted.

11.2.20. init-edge-paint-function-list

Arguments: none

Returns: nothing of significance

Side effects: All the edge-paint-functions are deleted (i.e., forgotten), except for the default-function.

Description: This function provides a means to (re-)initialize (or clear) the edge-paint-function precedence list. Deleting only particular edge-paint-functions from the precedence list entails initializing the list and then adding the ones that should not have been deleted.

11.2.21. init-font-function-list

Arguments: none

Returns: nothing of significance

Side effects: All the font-functions are deleted (i.e., forgotten), except for the default-function.

Description: This function provides a means to (re-)initialize (or clear) the font-function precedence list. Deleting only particular font-functions from the precedence list entails initializing the list and then adding the ones that should not have been deleted.

11.2.22. init-highlight-node-function-list

Arguments: none

Returns: nothing of significance

Side effects: All the highlight-node-functions are deleted (i.e., forgotten), except for the default-function.

Description: This function provides a means to (re-)initialize (or clear) the highlight-node-function precedence list. Deleting only particular highlight-node-functions from the precedence list entails initializing the list and then adding the ones that should not have been deleted.

11.2.23. init-node-paint-function-list

Arguments: none

Returns: nothing of significance

Side effects: All the node-paint-functions are deleted (i.e., forgotten), except for the default-function.

Description: This function provides a means to (re-)initialize (or clear) the node-paint-function precedence list. Deleting only particular node-paint-functions from the precedence list entails initializing the list and then adding the ones that should not have been deleted.

11.2.24. init-pname-function-list

Arguments: none

Returns: nothing of significance

Side effects: All the pname-functions are deleted (i.e., forgotten), except for the default-function.

Description: This function provides a means to (re-)initialize (or clear) the pname-function precedence list. Deleting only particular pname-functions from the precedence list entails initializing the list and then adding the ones that should not have been deleted.

11.2.25. init-pname-height-function-list

Arguments: none

Returns: nothing of significance

Side effects: All the pname-height-functions are deleted (i.e., forgotten), except for the default-function.

Description: This function provides a means to (re-)initialize (or clear) the pname-height-function precedence list. Deleting only particular pname-height-functions from the precedence list entails initializing the list and then adding the ones that should not have been deleted.

11.2.26. init-pname-length-function-list

Arguments: none

Returns: nothing of significance

Side effects: All the pname-length-functions are deleted (i.e., forgotten), except for the default-function.

Description: This function provides a means to (re-)initialize (or clear) the pname-length-function precedence list. Deleting only particular pname-length-functions from the precedence list entails initializing the list and then adding the ones that should not have been deleted.

11.2.27. init-unhighlight-node-function-list

Arguments: none

Returns: nothing of significance

Side effects: All the unhighlight-node-functions are deleted (i.e., forgotten), except for the default-function.

Description: This function provides a means to (re-)initialize (or clear) the unhighlight-node-function precedence list. Deleting only particular unhighlight-node-functions from the precedence list entails initializing the list and then adding the ones that should not have been deleted.

11.3. Global Variables

There are various global variables used by the ISI Grapher during normal operation, which an application built on top of the Grapher may want to inspect and (less often) modify. We now list some of these variables:

browser-window-record-list - a list of all known Grapher window objects. Each one of these objects contains various variables and structures associated with a particular Grapher window/graph.

active-browser-window-record - the currently active (or most recently active) Grapher window object, which also contains the values of many related variables.

grapher-io-window - the window in which interaction with the user takes place. During normal Grapher operation, various messages get printed to this window, and for certain commands, user input is read from this window.

graph-window - the currently active (or most recently active) Grapher window.

node-list - the list of nodes associated with the current graph. Each one of these objects has considerable structure, which is described elsewhere.

edge-list - the list of edges associated with the current graph. Each one of these objects has considerable structure, which is described elsewhere.

children-function - the name of the children-function used in the current graph.

command-menu-item-list - the list of items constituting the main command menu. To add to this list, use the function **add-to-command-menu**.

default-layout-style - the default layout style used by the Grapher if the Grapher is called with this value unspecified; one of **'tree** or **'lattice**.

font-list - the list of fonts which are available to the Grapher.

hash-table-size - the default size for hash tables created and used by the Grapher.

graph-layout-style - the layout style used in the current graph.

highlighted-node - the currently highlighted Grapher node, if any.

known-visible-edges - the edges currently visible (or partially visible) in the current window.

known-visible-nodes - the nodes currently visible (or partially visible) in the current window.

logical-x-displacement - the value added to all X coordinates (after multiplication by **x-stretch-factor**.)

logical-y-displacement - the value added to all Y coordinates (after multiplication by **y-stretch-factor**.)

object-height - an integer representing the height, in display units (pixels), of the entire graph.

object-width - an integer representing the width, in display units (pixels), of the entire graph.

parents-function - the name of the parents-function used in the current graph.

root-nodes - the list of root nodes of the current graph.

tracking-mouse - This variable if non-NIL if and only if the grapher is currently tracking the mouse. An alternate way of exiting the grapher is to set the global variable **tracking-mouse** to NIL. This is useful when an application running concurrently with the grapher wishes the grapher to exit out of the main mouse-tracking loop.

x-stretch-factor - the number by which all X-coordinates are multiplied.

y-stretch-factor - the number by which all Y-coordinates are multiplied.

The global variables listed above should not be carelessly modified by an application, as the normal operation of the Grapher depends on the correctness and consistency of the values associated with these variables; however, these variables may be freely inspected by an application. Many of these variables (and others) may be modified via special functions described elsewhere in this document. Their listing here is provided mainly as a convenience to the application builder.

11.4. Node and Edge Objects

Each node and edge in the graph is represented internally by the ISI Grapher as a separate object. In particular, each such object is represented by a Common-LISP *defstruct record* instance. Each such record instance contains numerous fields which the application-builder may inspect or (less often) modify. In this manual, whenever the argument(s) or the result of a function is listed as a *node* or an *edge*, the intention is that it is a node or edge record instance. We now summarize some of the functions which may be used to extract information from nodes and edges; keep in mind that the information returned by these functions is not computed dynamically when these functions are invoked, but rather is the stored result of earlier computations. In this sense the functions described here are merely simple accessor functions.

11.4.1. edge-already-visited-p

Argument: an edge

Returns: whatever value was stored in this slot before

Side effects: none

Description: This slot is very useful when doing traversals of searches of the graph over its (nodes and) edges. Typically, the searching function would select some unique value (such as a gensym or a unique integer), and start the search/traversal at the root-nodes of the graph, assigning that value to this slot as it visits each edge. When it encounters its own value in this slot, it knows that it has already visited (hence the name of this slot) this edge before, and can therefore skip processing it this time. This scheme allows arbitrary recursive graph traversals of the graph while insuring that each edge is traversed at most once, without having to keep a global record of such visits.

11.4.2. edge-containing-window

Argument: an edge

Returns: the window in which this edge is normally displayed

Side effects: none

Description: This function returns the window that contains the given edge

To modify any of the slots accessed by the above function calls, the Common LISP *setf* form may be used. For example, to set the pname of the node record instance N to "my graph node N", evaluate the form:

(setf (**node-pname** N) "my graph node N")

Similar syntax holds for assigning values to the slots of edge records instances. It is stressed that care must be exercised when modifying the values of the slots of Grapher record instances; it is up to the application-builder to make sure the resulting modification leaves these records in a consistent state with respect to the corresponding application.

11.4.3. edge-from-node

Argument: an edge

Returns: the node from which the given edge emanates

Side effects: none

Description: Edges are represented by the Grapher as ordered pairs of nodes. This function returns the first node of the ordered pair of nodes corresponding to the given edge. That is, it returns the node adjacent to this edge, and away from where the edge points.

11.4.4. edge-p

Argument: any LISP object

Returns: T if the argument is a Grapher edge object; otherwise NIL

Side effects: none

Description: This is a predicate that recognizes a Grapher edge object. Objects must pass this test (either explicitly, or implicitly) before they are submitted as arguments to functions which expect to receive a Grapher edge object.

11.4.5. edge-to-node

Argument: an edge

Returns: the node at which the given edge terminates

Side effects: none

Description: Edges are represented by the Grapher as ordered pairs of nodes. This function returns the second node of the ordered pair of nodes corresponding to the given edge. That is, it returns the node adjacent to this edge, and to which the edge points.

11.4.6. node-already-visited-p

Argument: a node

Returns: whatever value was previously stored in this slot before

Side effects: none

Description: This slot is very useful when doing traversals of searches of the graph via its nodes. Typically, the searching function would select some unique value (such as a gensym or a unique integer), and start the search/traversal at the root-nodes of the graph, assigning that value to this slot as it visits each node. When it encounters its own value in this slot, it knows that it has already visited (hence the name of this slot) this node before, and can therefore skip

processing it this time. This scheme allows arbitrary recursive graph traversals of the graph while insuring that each node is traversed at most once, without having to keep a global record of such visits.

11.4.7. node-children

Argument: a node

Returns: a list of nodes, representing the children of the given node with respect to the original graph.

Side effects: none

Description: This function returns a list of all the children nodes of the given node. Each one of these returned objects represents a child of the given node in the original graph.

11.4.8. node-containing-window

Argument: a node

Returns: the window in which this node is normally displayed

Side effects: none

Description: This function returns the window that contains the given node.

11.4.9. node-font

Argument: a node

Returns: the font that is used to display the pname

Side effects: none

Description: This function returns the font used to display the pname of the object represented by this Grapher node. This is the font assigned to this node earlier during the execution by the font-function.

11.4.10. node-group

Argument: a node

Returns: an inclusive list of nodes which also represent the object represented by the

given node.

Side effects: none

Description: This function returns a list of all the nodes which represent the same graph object, namely the object represented by the argument node. By definition, the argument node is included in this list. If the length of this list is greater than one, the **'tree** layout style must have been used to layout the graph. This is also the list of nodes all of which get highlighted when the mouse points to any one of them.

11.4.11. node-name

Argument: a node

Returns: the object represented by this given node instance.

Side effects: none

Description: This function returns the (application-dependent) object represented by this Grapher node instance. For example, if the node N is one created during the execution of the NIKL-Browser, (**node-name** N) would return a NIKL concept.

11.4.12. node-p

Argument: any LISP object

Returns: T if the argument is a Grapher node object; otherwise NIL

Side effects: none

Description: This is a predicate which recognizes a Grapher node object. Objects must pass this test (either explicitly, or implicitly) before they are submitted as arguments to functions which expect to receive a Grapher node object.

11.4.13. node-parents

Argument: a node

Returns: a list of nodes, representing the parents of the given node with respect to the original graph

Side effects: none

Description: This function returns a list of all the parent nodes of the given node. Each one

of these returned objects represents a parent of the given node in the original graph. A graph is a tree if and only if each node has at most one parent.

11.4.14. node-pname

Argument: a node

Returns: the string corresponding to the print-name of the object represented by this given node object

Side effects: none

Description: This function returns the print-name of the object represented by this Grapher node object. The print-name of this object is initially determined by the pname-function as discussed earlier. For example, if the node N is one created during the execution of the NIKL-Browser, (**node-pname** N) would return the print-name of the corresponding NIKL concept.

11.4.15. node-pname-height

Argument: a node

Returns: an integer, corresponding to the height (in pixels) of the pname associated with this node

Side effects: none

Description: This function returns the height of the pname used to display the given node. This is the pname-height assigned to this node earlier during the execution by the pname-height-function.

11.4.16. node-pname-length

Argument: a node

Returns: an integer, corresponding to the length (in pixels) of the pname associated with this node

Side effects: none

Description: This function returns the length of the pname used to display the given node. This is the pname-length assigned to this node earlier during the execution by the pname-length-function.

11.4.17. node-x-coordinate

Argument: a node

Returns: an integer, corresponding to the X coordinate (in pixels) of the given node, with respect to the layout

Side effects: none

Description: This function returns the X coordinate, in absolute coordinates, of the given node, with respect to the current layout computed earlier in the execution. This is the X coordinate assigned to this node by **layout-x-and-y**.

11.4.18. node-y-coordinate

Argument: a node

Returns: an integer, corresponding to the Y coordinate (in pixels) of the given node

Side effects: none

Description: This function returns the Y coordinate, in absolute coordinates, of the given node, with respect to the current layout computed earlier in the execution. This is the Y coordinate assigned to this node by **layout-x-and-y**.

11.5. Other Useful Functions

We now list various useful functions which an application-builder might wish to utilize:

11.5.1. add-node-to-graph

Argument 1: parent node or list of parent nodes

Argument 2: name of new node (a string)

Argument 3: dont-re-layout flag (optional)

Argument 4: dont-redraw flag (optional)

Returns: nothing of significance

Side effects: A new node is created, having the given name. It is then added to the current graph as the son of the given parent node(s). The various data structures of the graph are updated to reflect this change.

Description: This function is used to add a new node to an already-existing graph. If the

dont-re-layout flag is non-NIL, no relayout will take place after the addition of the new node (this is useful to do when several nodes are to be added, as relayout is an time-consuming operation; relayout may then be done after the last node has been added by calling the function **relayout-x-and-y**, or by setting the said flag to NIL on the last call to **add-node-to-graph**.) If the "dont-redraw" flag is non-NIL, no redrawing will take place after the addition of the new node, and the previous remarks hold for this flag also.

11.5.2. bury-windows

Arguments: none

Returns: nothing of significance

Side effects: A menu is displayed containing all existing grapher windows, and the user is requested to select the one which he would like to become buried. The selected window is then buried.

Description: This function is intended to provide a convenient means for the user to keep track of which grapher windows exists and also to quickly bury windows. This function is also accessible from the main command menu.

11.5.3. center-this-node

Argument 1: a node

Argument 2: a window (optional)

Argument 3: dont-redraw flag (optional)

Returns: nothing of significance

Side effects: The grapher centers the current window around the given node; that is, the current window is then automatically scrolled so that the given node becomes visible and centered in that window.

Description: This function is sometimes called by **find-named-node**. It is used to quickly "jump" to arbitrary nodes. If dont-redraw-flag is non-NIL, no redrawing will take place after the scroll. This option is provided when several other operations are to be performed before any redrawing should take place.

11.5.4. delete-from-command-menu

Argument : string-line

Returns: nothing of significance

Side effects: The command/menu-line with the given description is removed from the main command menu.

Description: This function is used to delete a menu item from the main command menu. This function does the opposite of **add-to-command-menu**. The argument should correspond exactly to the string associated from the said command.

11.5.5. delete-node-from-graph

Argument 1: a node

Argument 2: dont-re-layout-flag (optional)

Argument 3: dont-redraw-flag (optional)

Returns: nothing of significance

Side effects: The given node is removed from the current graph. The various data structures are updated to reflect this change. All nodes which become parentless as a result of this deletion are also deleted.

Description: This function is used to delete a node from an already-existing graph. If dont-re-layout-flag is non-NIL, no relayout will take place after the deletion of the node (this is useful when several nodes are to be deleted, as relayout is an time-consuming operation; relayout may then be done after the last node has been deleted by calling the function **relayout-x-and-y**, or by setting the said flag to NIL on the last call to **delete-node-from-graph**.) If dont-redraw-flag is non-NIL, no redrawing will take place after the deletion of the node, and the previous remarks hold for this flag also. Note that all nodes that become parentless as a result of the original deletion are also deleted, and this rule is applied recursively to their children. To delete *all* the descendents of a given node (not just the ones that become parentless), use the function **delete-subtree-from-graph**. Thus, for trees, the functions **delete-node-from-graph** and **delete-subtree-from-graph** have the same semantics.

11.5.6. delete-subtree-from-graph

Argument 1: a node

Argument 2: dont-re-layout-flag (optional)

Argument 3: dont-redraw-flag (optional)

Returns: nothing of significance

Side effects: The given node *and* the entire subtree rooted at this node is removed from the current graph. The various data structures are updated to reflect this change.

Description: This function is used to delete a subtree from an existing graph. If dont-re-layout-flag is non-NIL, no relayout will take place after the deletion of the subtree (this is useful when several subtrees are to be deleted, as relayout is an time-consuming operation; relayout may then be done after the last subtree has been deleted by calling the function **relayout-x-and-y**, or by setting the said flag to NIL on the last call to **delete-subtree-from-graph**.) If dont-redraw-flag is non-NIL, no redrawing will take place after the deletion of the subtree, and the previous remarks hold for this flag also. For trees, the functions **delete-node-from-graph** and **delete-subtree-from-graph** have the same semantics, but generally for lattices, **delete-subtree-from-graph** will remove more nodes from a given graph than **delete-node-from-graph**.

11.5.7. displace-object

Argument 1: x displacement (an integer)

Argument 2: y displacement (an integer)

Returns: nothing of significance

Side effects: The entire graph (in the current graph window) is displaced from the origin by the corresponding given x and y displacements.

Description: The logical origin for the graph becomes the given x and y values, respectively. These values are taken into account whenever coordinates of nodes and edges are subsequently calculated.

11.5.8. displaced-edge-x1

Argument: an edge

Returns: the x coordinate of the node from which the given edge is directed

Side effects: none

Description: This function returns the x coordinate of the "from" node of the given edge, taking into account any logical displacement and a possible stretch factor that may be in effect. This function should be used, rather than peaking directly into the corresponding slot, which contains the relevant value in absolute coordinates.

11.5.9. displaced-edge-x2

Argument: an edge

Returns: the x coordinate of the node toward which the given edge is directed

Side effects: none

Description: This function returns the x coordinate of the "to" node of the given edge, taking into account any logical displacement and a possible stretch factor that may be in effect. This function should be used, rather than peaking directly into the corresponding slot, which contains the relevant value in absolute coordinates.

11.5.10. displaced-edge-y1

Argument: an edge

Returns: the y coordinate of the node from which the given edge is directed

Side effects: none

Description: This function returns the y coordinate of the "from" node of the given edge, taking into account any logical displacement and a possible stretch factor that may be in effect. This function should be used, rather than peaking directly into the corresponding slot, which contains the relevant value in absolute coordinates.

11.5.11. displaced-edge-y2

Argument: an edge

Returns: the y coordinate of the node toward which the given edge is directed

Side effects: none

Description: This function returns the y coordinate of the "to" node of the given edge, taking into account any logical displacement and a possible stretch factor that may be in effect. This function should be used, rather than peaking directly into the corresponding slot, which contains the relevant value in absolute coordinates.

11.5.12. displaced-node-x-coordinate

Argument: a node

Returns: the x coordinate of the given node

Side effects: none

Description: This function returns the x coordinate of the given node, taking into account any logical displacement and a possible stretch factor that may be in effect. This function should be used to determine the x coordinate of a node, rather than peaking directly into the x-coordinate node slot, which contains the relevant value in absolute coordinates.

11.5.13. displaced-node-y-coordinate

Argument: a node

Returns: the y coordinate of the given node

Side effects: none

Description: This function returns the y coordinate of the given node, taking into account any logical displacement and a possible stretch factor that may be in effect. This function should be used to determine the y coordinate of a node, rather than peaking directly into the y-coordinate node slot, which contains the relevant value in absolute coordinates.

11.5.14. expose-windows

Arguments: none

Returns: nothing of significance

Side effects: A menu is displayed containing all existing grapher windows, and the user is requested to select the one to be exposed. The selected window then becomes the currently active grapher window.

Description: This function is intended to provide a convenient means for the user to keep track of which grapher windows exist and also to quickly expose buried windows. This function is also accessible from the main command menu.

11.5.15. find-central-node

Argument: a list of nodes

Returns: the node closest to the "center of mass" of the nodes in the given list

Side effects: none

Description: This function returns the node that is most "central" among the nodes in the given node list. That is, the node closest to the point (**x**,**y**) is returned, where **x** is the average of the x coordinates of the nodes in the node list, and **y** is the average of the y coordinates of the nodes in the node list. This function is used by the grapher in various situations to determine the node around which the graph window should be centered.

11.5.16. find-named-node

Argument: the name of a node

Returns: nothing of significance

Side effects: The grapher tries to find a node with the given name in the current window; the current window is then automatically scrolled so that the named node becomes visible and centered in that window. If the named node can not be found in the current window, other grapher windows are also searched, until such a node is found. If the search is still unsuccessful, it is repeated, this time without regard to case. If a node with such a name is nowhere to be found, an error message is printed.

Description: This function is sometimes called by **find-node**. It is used to quickly "jump" to arbitrary nodes simply by naming them.

11.5.17. find-node

Argument: a node

Returns: nothing of significance

Side effects: A menu is displayed allowing the user to directly "go to" the given node, any of its parents or children, or he can type in the name of an arbitrary node that he would like to go to. The current display is automatically scrolled so that the said node is visible and centered in the window.

Description: This function is also directly accessible from the main command menu. It is used to quickly "jump" to arbitrary nodes without having to manually search for them or slowly scroll to them.

11.5.18. global-scroll

Argument: a window

Returns: nothing of significance

Side effects: global scrolling is initiated in the given grapher window

Description: This function provides global scrolling capability in a given grapher window. A small global-map window appears alongside the current grapher window, and the user may then quickly scroll to any part of the graph via mouse motion in the corresponding direction. The global-map window contains a miniature picture of the entire graph, scaled to fit in that window; a dark box highlights the part of the graph that is currently visible in the graph window, and both windows are dynamically updated to reflect the changing position in the graph as controlled by the user via the mouse. This function is also called by **scroll** when the global-scroll option is selected.

11.5.19. grapher-hard-copy

Argument 1: a window (optional)

Argument 2: a file name (optional)

Returns: nothing of significance

Side effects: The graph in the current window is converted to a series of bitmaps, each of size less than or equal to the size of the current window. Each such bitmap is written to a separate file (whose prefix is the given file name). It is then up to the user to hardcopy the bitmap in the resulting files using his favorite bitmap-hardcopying function.

Description: Global scrolling occurs in the window (with a small margin of overlap), and each window-full is converted to a bitmap and gets written to a separate file. Its the user responsibility to later actually hardcopy the resulting bitmaps using a (system-dependent) hardcopy function. Such a function cannot be included as part of the grapher because it is too implementation and system dependent; any such attempt is bound to greatly reduce the portability of the entire grapher. This functionality is merely meant to be a convenient and automatic mechanism to produce a "mosaic" from the displayed graph. In addition, on systems where a certain bitmap-creating function is missing, this function cannot be executed. We hope the entire process of hardcopying will be made less painful in future releases. This function is also called by **scroll** when the global-scroll option is selected.

11.5.20. highlight group

Argument 1: a node

Argument 2: window (optional)

Returns: nothing of significance

Side effects: The given node, as well as all of the nodes identified with it (via the cycle-elimination schema described elsewhere in this document), are highlighted in the current graph window.

Description: This is the main highlighting routine that is called whenever a node and its "highlight-group" are to be highlighted. For example, this function is invoked whenever the mouse enters the screen area associated with a node in the currently active grapher window.

11.5.21. information

Arguments: none

Returns: nothing of significance

Side effects: Various useful information and statistics about the grapher and the current graph window are displayed.

Description: This function is intended to provide a convenient means for the user to obtain certain help and information about the grapher and the current graph window. This function is also accessible from the main command menu.

11.5.22. kill-all-windows

Arguments: none

Returns: nothing of interest.

Side effects: All known window records are permanently removed from **browser-window-record-list**, the associated windows are killed, and the associated graphs and related data structures are disposed of. This is accomplished by individual calls to **kill-window-record** once for each window record.

Description: This is the function called from the main command menu to kill all the Grapher windows.

11.5.23. kill-window-record

Argument: a Grapher window record

Returns: nothing of interest.

Side effects: The given window record is permanently removed from **browser-window-record-list**, the associated window is killed, and the associated graph and related data structures are disposed of.

Description: This is the function called from the main command menu to kill a Grapher window.

11.5.24. kill-windows

Arguments: none

Returns: nothing of significance

Side effects: A menu is displayed containing all existing grapher windows, and the user is requested to select the one to be killed. The selected window is then killed.

Description: This function is intended to provide a convenient means for the user to keep track of which grapher windows exists and also to permanently kill windows. This function is also accessible from the main command menu. Note that when a window is killed, its graph and associated data structures are reclaimed and thus unretrievable.

11.5.25. layout-x-and-y

Arguments: none

Returns: nothing-of-significance

Side effects: The current graph is (re)layed-out; the coordinates of all nodes are (re)computed.

Description: This function is the main graph layout function. It computes appropriate coordinates for all the nodes in the graph. The time this operation takes is proportional to the size of the graph. All the relevant graph data structures are assumed to already exist. This operation should be done at the end of any series of modifications to the graph (such as node or edge deletions or additions).

11.5.26. local-scroll

Argument 1: a scroll command

Argument 2: a window

Returns: nothing of significance

Side effects: Scrolling of the given grapher window is initiated, according to the given scroll command.

Description: This function provides local scrolling capability in a given grapher window.

The scroll command argument may be one of the symbols {local-up, local-down, local-left, local-right, local-right-and-up, local-right-and-down, local-left-and-up, local-left-and-down, local-go-to-center}. Each one of these causes scrolling of one screen-full in the corresponding direction, while the last one causes the window to become centered on the node closest to the "center" of the graph. This function is also called by **scroll** when the local-scroll option is selected.

11.5.27. move-node-in-graph

Argument 1: to-node

Argument 2: move-node

Argument 3: dont-re-layout flag (optional)

Argument 4: dont-redraw flag (optional)

Returns: nothing-of-significance

Side effects: The move-node and its associated subtree is moved in the graph to become the son of the to-node.

Description: This function moves an entire subtree from one place in the graph to another. The move-node argument specifies which node is to be moved, while the to-node argument specifies the node below which this subgraph will be transplanted. The optional dont-relayout and dont-redraw flags, if non-NIL, suppress relaying-out and redrawing of the graph, respectively; this is useful when such operations are only desired at the end of a long sequence of modifications to the graph (for efficiency reasons.)

11.5.28. redraw

Argument: a window

Returns: nothing-of-significance

Side effects: The given graph window is cleared and redrawn; various parameters are recomputed (if necessary) for faster future redraws.

Description: This function exposes and clears the given window, and then the associated graph is redrawn. This function is typically invoked after some operation has modified the appearance of the graph (such as stretching or node adding, etc.) It is also accessible from the main command menu.

11.5.29. save-global-variables

Argument: a browser window record

Returns: nothing-of-significance

Side effects: The global variables of the current graph are saved in the given browser window record. This is used when control in the Grapher is transferred between one graph/window and another.

Description: This function records the values of all the relevant global variables associated with the current graph into the given record. The record of the currently active graph is given by the global variable **active-browser-window-record**. This function is typically invoked before control transfers to another grapher window/graph. The counterpart function is **set-global-variables**.

11.5.30. scroll

Argument: a window

Returns: nothing of significance

Side effects: A menu is displayed containing several scrolling methods, and the user is requested to select one. The corresponding scrolling mode then commences in the given window.

Description: This function is intended to provide a convenient interface to the various scrolling mechanisms. This function is also accessible from the main command menu.

11.5.31. set-global-variables

Argument: a browser window record

Returns: nothing-of-significance

Side effects: The global variables of the current graph are restored from the given browser window record. This is used when control in the Grapher is transferred between one graph/window and another.

Description: This function restores (overwrites) the values of all the relevant global variables associated with the current graph from the corresponding values in the given record. The record of the currently active graph is given by the global variable **active-browser-window-record**. This function is typically invoked after control transfers to another grapher window/graph. The counterpart function is **save-global-variables**.

11.5.32. set-up-defaults

Arguments: none

Returns: nothing of significance

Side effects: initializes all the user-modifiable functions to their default values

Description: This function is equivalent to executing each one of the following functions: init-pname-function-list, init-pname-length-function-list, init-pname-height-function-list, init-highlight-node-function-list, init-unhighlight-node-function-list, init-describe-function-list, init-node-paint-function-list, and init-edge-paint-function-list.

11.5.33. track-the-mouse

Arguments: none

Returns: nothing of significance

Side effects: The current grapher window is exposed and grapher-related mouse-tracking begins.

Description: This function tracks the mouse in the current Grapher window and waits for mouse events to transpire. All the commands as well as other user-interactions are available and are activated from this function. Graph nodes are highlighted as appropriate when selected by the mouse. This function is exited when the user clicks outside any grapher window or selects the "exit" option from the main command menu. An alternate way of exiting this function (and thus the grapher) is to set the global variable **tracking-mouse** to NIL. This function is the main mouse-tracking function used by the grapher.

11.5.34. un-highlight-group

Argument 1: a node

Argument 2: window (optional)

Returns: nothing of significance

Side effects: The given node, as well as all of the nodes which are identified with it (via the cycle-elimination schema described elsewhere in this document) are unhighlighted in the current graph window.

Description: This is the main unhighlighting routine that is called whenever a node and its "highlight-group" are to be unhighlighted. For example, this function is invoked whenever the mouse exits the screen area associated with a node in the currently active grapher window.

11.6. Utility Functions

11.6.1. unique-integer

Argument: none

Returns: a unique integer (an integer that was never returned from this function by a previous call).

Side effects: none

Description: This function returns a large unique integer. All integers returned by subsequent calls to this function are guaranteed to be different. This is useful when a search or a traversal of the graph is conducted via marking the node-already-visited-p (and edge-already-visited-p) field(s). The mark may be a unique integer, which will distinguish it from marks left by previous searches.

11.6.2. browser-print

Argument 1: message

Argument 2: spacing (optional)

Argument 3: stay (optional)

Argument 4: stream (optional)

Argument 3: clear (optional)

Returns: nothing of significance

Side effects: This function prints a message to the given (or a default) input/output stream.

Description: This function is used to print error messages and other informative messages that the grapher generates. The argument "message" is the message text string. The optional argument spacing is an integer telling how many lines to skip before printing the message text, with a default of single spacing. The optional flag "stay", if non-NIL, instructs this function to leave the message window exposed until the user presses any key to continue. This is useful if

the user is likely to need more time to inspect the message. The optional argument "stream," if present, will override the default IO stream. The flag "clear", if non-NIL, will cause the message window to be cleared before the message will be printed.

11.6.3. browser-read

Argument 1: message

Argument 2: window (optional)

Argument 3: spacing (optional)

Returns: a value read from the keyboard.

Side effects: This function prints a message to the given (or a default) input/output stream, and then reads some value provided by the user.

Description: The argument "message" is an explanatory text string, to be printed before any reading is attempted. The optional argument "window," if present, overrides the default IO stream. The optional argument "spacing," if present, controls how many lines will be skipped before printing the message text, with a default of single spacing.

11.6.4. browser-read-string

Argument 1: message

Argument 2: window (optional)

Argument 3: spacing (optional)

Returns: a string read from the keyboard.

Side effects: This function prints a message to the given (or a default) input/output stream, and then reads some string typed by the user.

Description: The argument "message" is an explanatory text string, to be printed before any reading is attempted. The optional argument "window," if present, overrides the default IO stream. The optional argument "spacing," if present, controls how many lines will be skipped before printing the message text, with a default of single spacing. This function is very similar to the function browser-read, except that the user input is interpreted as a string instead of as an arbitrary value.

11.6.5. draw-box

Argument 1: x position

Argument 2: y position

Argument 3: length

Argument 4: height

Argument 5: window

Argument 6: mode

Returns: nothing of significance

Side effects: A box is drawn to the given window at the specified position, having the specified dimensions, and using the given drawing mode.

Description: The position in the window is given by x and y, where (0,0) is the upper left-hand corner of the window, while the X and Y axis increase to the left and down, respectively. The length and width of the box are assumed to be in pixels, and "mode" may be one of 'xor, 'erase, or 'overwrite.

11.6.6. inside-node-p

Argument 1: a node

Argument 2: x coordinate

Argument 3: y coordinate

Returns: non-NIL if the given point falls inside the given node; otherwise, NIL

Side effects: none

Description: This predicate determines whether the given point, specified by its x and y coordinates, falls inside the area associated with the given node; non-NIL is returned if and only if this is the case.

11.6.7. ldifference

Argument 1: a list

Argument 2: a list

Returns: a list representing the set difference between the two lists

Side effects: none

Description: This function returns a list corresponding to the list of items that appear on the first list, but not on the second list. This function is hacked for efficiency: if the lists are longer than a certain fixed size, a hash-table is used to compute the result.

11.6.8. make-browser-hash-table

Argument: size (optional)

Returns: a hash table

Side effects: creates and returns a hash table of the specified size, if the size is not specified, a default size is selected. If this hash table becomes almost full at any time in the future, it is automatically replaced by a larger hash table.

Description: This is the function called by the grapher to create hash tables.

11.6.9. remove-duplicate-nodes

Argument: a list of nodes

Returns: the same list except all duplicate nodes are removed

Side effects: none

Description: This function removes all the duplicate nodes from the given list and returns the result. It sorts the list first, then uses a linear scan, so the time complexity is O(NlogN) where N is the length of the given list (as opposed to quadratic time using the naive algorithm; a hash table would make the time almost linear, but creating a hash table is expensive by itself).

11.6.10. set-if-not-bound

Argument 1: a quoted variable name

Argument 2: an object to be evaluated

Returns: nothing of significance

Side effects: If the given variable is not already bound, or is bound to NIL, the second argument is evaluated and the resulting value is bound to that variable name.

Description: This function is very much like the standard Common LISP "set" function, except that if the given variable is already bound to some non-NIL value, no action is taken and the entire call has no effect.

11.6.11. transitive-closure

Argument: a node

Returns: the transitive closure of the graph starting at the given node.

Side effects: none

Description: This function returns all the graph nodes that are reachable from the given node (via directed paths).

11.7. Mapping Functions

The following set of orthogonal functions provide conversion between object names, graph nodes, and graph sons and parents. That is, given an object name, the corresponding graph node can be obtained, etc. These conversions are summarized in the following table:

Function	Input	Output
name-to-node	node name	node
name-to-son-nodes	node name	son nodes
name-to-son-names	node name	son names
node-to-son-nodes	node	son nodes
node-to-son-names	node	son names
name-to-parent-nodes	node name	parent nodes
name-to-parent-names	node name	parent names
node-to-parent-nodes	node	parent nodes
node-to-parent-names	node	parent names

Not all of these functions are currently used by the Grapher, but they are provided as an interface mechanism to applications that are built on top of the Grapher. The following sections describe each of these functions.

11.7.1. name-to-node

Argument: a LISP object

Returns: the graph node corresponding to the given object

Side effects: none

Description: This function returns the graph node (from the currently active graph) associated with the specified object. If no graph node is associated with the given object, NIL is returned.

11.7.2. name-to-parent-names

Argument: a LISP object

Returns: the objects associated with the parents of the graph node corresponding to the given object

Side effects: none

Description: This function returns the objects corresponding to the parents of the graph node (from the currently active graph) associated with the specified object. If no graph node is associated with the given object, NIL is returned.

11.7.3. name-to-parent-nodes

Argument: a LISP object

Returns: the parent (nodes) of the graph node corresponding to the given object

Side effects: none

Description: This function returns the parents of the graph node (from the currently active graph) associated with the specified object. If no graph node is associated with the given object, NIL is returned.

11.7.4. name-to-son-names

Argument: a LISP object

Returns: the objects associated with the children of the graph node corresponding to the given object

Side effects: none

Description: This function returns the objects corresponding to the children of the graph node (from the currently active graph) associated with the specified object. If no graph node is associated with the given object, NIL is returned.

11.7.5. name-to-son-nodes

Argument: a LISP object

Returns: the children (nodes) of the graph node corresponding to the given object

Side effects: none

Description: This function returns the children of the graph node (from the currently active graph) associated with the specified object. If no graph node is associated with the given object, NIL is returned.

11.7.6. node-to-parent-names

Argument: a node

Returns: the names of the parents of the given graph node

Side effects: none

Description: This function returns the list of objects associated with the parents of the given graph node (from the currently active graph).

11.7.7. node-to-parent-nodes

Argument: a node

Returns: the parent (nodes) of the given graph node

Side effects: none

Description: This function returns the list of parent nodes of the given graph node (from the currently active graph).

11.7.8. node-to-son-names

Argument: a node

Returns: the names of the children of the given graph node

Side effects: none

Description: This function returns the list of objects associated with the children of the given graph node (from the currently active graph).

11.7.9. node-to-son-nodes

Argument: a node

Returns: the children (nodes) of the given graph node

Side effects: none

Description: This function returns the list of children nodes of the given graph node (from the currently active graph).

11.8. Implementation-dependent Functions

This section described various implementation-dependent functions. These functions are the primary candidates to change when a port to a new system is undertaken. Functions in other sections are less likely to require modification because they are written in Common LISP. Many functions use the functions in this section as primitives; for example, **draw-box** makes four calls to **draw-line** in order to draw a box on the display. The former is implementation independent while the latter is not.

11.8.1. bold-font

Argument: none

Returns: the font which is currently considered the bold font

Side effects: The default font of the currently active grapher window is set to the bold font.

Description: This function is used to change the default font of the current graph window to the "bold font." The value of the bold font is user controlled, and is initially set to a reasonable default.

11.8.2. bury-window

Argument: window

Returns: the given window

Side effects: The given window is buried

Description: This is the standard function that buries a given window. That is, the given window is de-exposed, and sent to the back of the window stack, usually

completely disappearing from view.

11.8.3. clear-window

Argument: window

Returns: the given window

Side effects: The given window is cleared.

Description: This is the standard function that clears a given window. All previously displayed text and graphics in the window are erased.

11.8.4. de-expose-window

Argument: window

Returns: the given window

Side effects: The given window is de-exposed.

Description: This is the standard function that de-exposes a given window.

11.8.5. draw-circle

Argument 1: window

Argument 2: x

Argument 3: y

Argument 4: radius

Argument 5: mode (optional)

Returns: nothing of significance

Side effects: A circle with the given center and radius is drawn inside the given window, using the specified mode.

Description: A circle of the given radius is drawn at (x,y) within the given window. The mode may be one of **'xor**, **'erase**, or **'overwrite**. The default mode is **'overwrite**.

11.8.6. draw-line

Argument 1: window

Argument 2: x_1

Argument 3: y_1

Argument 4: x_2

Argument 5: y_2

Argument 6: mode (optional)

Returns: **'invisible** if neither end-point of the specified line lies within the given window; otherwise, **'visible**

Side effects: A line with the given end-points is drawn inside the given window, using the specified mode.

Description: A thin line is drawn from (x_1,y_1) to (x_2,y_2) within the given window. The mode may be one of **'xor**, **'erase**, or **'overwrite**. If either of the line end-points lie inside the window, **'visible** is returned, otherwise **'invisible** is returned. The default mode is **'overwrite**.

11.8.7. draw-rectangle

Argument 1: window

Argument 2: width

Argument 3: height

Argument 4: x

Argument 5: y

Argument 6: mode (optional)

Returns: nothing of significance

Side effects: A filled rectangle is drawn inside the given window, using the specified mode.

Description: A filled rectangle is drawn inside the given window, where the upper left corner of the rectangle is identified with location (x,y) within the window. The mode may be one of **'xor**, **'erase**, or **'overwrite**.

11.8.8. draw-string

Argument 1: a string

Argument 2: length

Argument 3: height

Argument 4: x

Argument 5: y

Argument 6: font

Argument 7: window

Returns: **'visible** if any part of the string is visible in the window; otherwise, **'invisible**

Side effects: prints the string in the specified window, according to the given parameters

Description: First a rectangle of the given length and height is erased in the window. The left upper corner of the rectangle is taken to be at the given x and y coordinates. This is used to clear an area before printing; for example, this is how strings overwrite other graphics without causing a cluttered-looking display (if no such erasing is desired, make the erase rectangle of size 0.) Next, the given string is printed in the window using the given font. If any part of the string is visible in the window after the printing, the atom **'visible** is returned. If none of the string appears within the window after the printing, the atom **'invisible** is returned. This is used to assist the calling function in recording information that would speed up future redraws (for example, if a string is outside the window's boundaries, and no scrolling or window-resizing has been done, future redraws would not attempt to draw this string.)

11.8.9. expose-window

Argument: window

Returns: the given window

Side effects: exposes the given window, if it is not already exposed

Description: This is the standard function that exposes a given window. That is, the given window is made visible. If the window is already exposed, no action is taken.

11.8.10. exposed-p

Argument: window

Returns: non-NIL if the given window is exposed, or NIL otherwise

Side effects: none

Description: This is the standard predicate that tests whether a given window is currently exposed.

11.8.11. font-pixel-height

Argument: a font

Returns: the height of the given font, in pixels

Side effects: none

Description: This function returns the character height of the given font, specified in pixels. It is used in drawing strings in windows.

11.8.12. font-pixel-width

Argument: a font

Returns: the width of the given font, in pixels

Side effects: none

Description: This function returns the character width of the given font, specified in pixels. If the given font is variable-width, it returns an "average" width for that font. It is used in drawing strings in windows.

11.8.13. get-all-fonts

Argument: none

Returns: all the available fonts in the system

Side effects: none

Description: This function returns the list of fonts available in the system. In subsequent grapher operations any one of these fonts may be used in drawing the print-names of nodes. The main menu provides a means for the user to change fonts

dynamically. In future releases, the idea of fonts may have to be generalized, as some systems do not support the notion of a font (such as Symbolics, Version 7.0).

11.8.14. get-changed-mouse-state

Arguments: none

Returns: the current state of the mouse, after it has changed

Side effects: the system waits for the mouse to move or a mouse key to be depressed.

Description: This function waits for a change in the state of the mouse (that is, a motion or a mouse-key action), and then returns the current mouse, as a multiple value list, which includes the x and y coordinates of the mouse cursor, as well as a value indicating which mouse key is being pressed, if any. This function is similar to **get-current-mouse-state**, except that it first waits until there is some change in the state of the mouse first.

11.8.15. get-current-mouse-state

Arguments: none

Returns: the current state of the mouse

Side effects: none

Description: This function returns the current mouse, as a multiple value list, which includes the x and y coordinates of the mouse cursor, as well as a value indicating which mouse key is being pressed, if any.

11.8.16. get-real-time

Argument: none

Returns: the number of milliseconds elapsed since the last boot

Side effects: none

Description: This function is used by the function unique-integer to help generate a unique integer; this integer is used as a flag during graph traversals. When porting the Grapher, the returned value here does not have to be in milliseconds; any unit corresponding to a small fraction of a second would suffice.

11.8.17. giant-font

Argument: none

Returns: the font that is currently considered the giant font.

Side effects: The default font of the currently active grapher window is set to the giant font.

Description: This function is used to change the default font of the current graph window to the "giant font." The value of the giant font is user-controlled, and is initially set to a reasonable default.

11.8.18. grapher-restart

Argument: (one argument, which is ignored)

Returns: the currently active graph window

Side effects: The currently active graph window is exposed and redrawn, and the grapher begins to track the mouse in that window.

Description: This function is an alternate high-level interface to the grapher, once a graph window exists; for example, this is the function that gets called when the user reactivates the grapher after it was suspended. If no graph-window exists, an appropriate error message will be generated.

11.8.19. italic-font

Argument: none

Returns: the font which is currently considered the italic font

Side effects: The default font of the currently active grapher window is set to the italic font.

Description: This function is used to change the default font of the current graph window to the "italic font." The value of the italic font is user-controlled, and is initially set to a reasonable default.

11.8.20. kill-window

Argument: window

Returns: the given window

Side effects: The given window is killed.

Description: This is the standard function that buries a given window. The window is de-exposed, and is permanently deleted.

11.8.21. make-browser-window

Argument 1: height (optional)

Argument 2: width (optional)

Argument 3: position (optional)

Returns: a newly created window

Side effects: A new window is created.

Description: This function creates and returns a new window. Windows are assumed to have the implicit ability to display text and line segments in arbitrary position, as well as the ability to be moved, (re)sized, cleared, exposed, etc. All reasonable window systems should support these operations. If any of the arguments to this function are missing, reasonable default values will be chosen.

11.8.22. menu-create

Argument 1: label

Argument 2: item-list

Argument 3: borders

Returns: a menu with the specified attributes

Side effects: A new menu object, having the given attributes, is created and returned.

Description: A menu having the given borders (thickness in pixels), label (a string), and menu items is created and returned. The item-list is a list of menu items each having the following form: **(menu-line documentation-line returned-value)**, where **menu-line** is the text to appear inside the menu for that item, **documentation-line** is the text to appear in the mouse-documentation part of the display (if any) once the corresponding menu item is highlighted, and **returned-value** is the value to be returned from the menu once the corresponding item has been selected. To actually pop-up the menu and solicit a user selection, the function **menu-select** should be used.

11.8.23. menu-select

Argument: a menu

Returns: the return-value associated with the menu-item selected by the user

Side effects: The given menu is exposed and activated, and the user is requested to select some item from the menu.

Description: Th given menu (previously created with the function **menu-create**) is exposed and activated. The user is then required to select some item from the menu via mouse interaction. When the various items of the menu are highlighted, their corresponding documentation-lines are displayed at the appropriate screen location. If the user selects a particular menu item, the corresponding returned-value is returned by this function; otherwise, NIL is returned.

11.8.24. normal-font

Argument: none

Returns: the font which is currently considered the normal font

Side effects: The default font of the currently active grapher window is set to the normal font.

Description: This function is used to change the default font of the current graph window to the "normal font." The value of the normal font is user-controlled, and is initially set to a reasonable default.

11.8.25. run-browser

Argument: dont-track flag

Returns: nothing of significance

Side effects: the currently active graph window is exposed and redrawn, and if the dont-track flag is NIL (or missing), the grapher begins to track the mouse in that window.

Description: This function is the highest-level interface to the grapher, once a graph window already exists. If the dont-track flag is non-NIL, the active graph window will still be exposed and redrawn but no mouse-tracking will be initiated. This is provided for flexibility reasons. A previously created graph window is assumed to already exist prior to the invocation of this function.

11.8.26. set-window-height

Argument 1: a window

Argument 2: new window height

Returns: nothing of significance

Side effects: The height of the given window is changed to the specified value.

Description: This function sets the height of the given window to the specified value. This is used to resize a window.

11.8.27. set-window-position

Argument 1: window

Argument 2: new x position

Argument 3: new y position

Returns: nothing of significance

Side effects: The given window is moved to a new location.

Description: The window is moved so that its upper left corner is at the given position (in pixels) on the display.

11.8.28. set-window-size

Argument 1: window

Argument 2: width

Argument 3: height

Returns: nothing of significance

Side effects: The given window is resized to the specified width and length.

Description: The size of the specified window is changed to become the new width and length (in pixels).

11.8.29. set-window-width

Argument 1: a window

Argument 2: new window width

Returns: nothing of significance

Side effects: The width of the given window is changed to the specified value.

Description: This function sets the width of the given window to the specified value. This is used to resize a window.

11.8.30. set-window-x

Argument 1: a window

Argument 2: new x position

Returns: nothing of significance

Side effects: The x position of the given window is changed to the specified value.

Description: This function sets the screen x-position of the upper-left corner of the given window to the specified value. This is used to move a window on the screen.

11.8.31. set-window-y

Argument 1: a window

Argument 2: new y position

Returns: nothing of significance

Side effects: The y position of the given window is changed to the specified value.

Description: This function sets the screen y-position of the upper-left corner of the given window to the specified value. This is used to move a window on the screen.

11.8.32. window-height

Argument: a window

Returns: the height of the given window

Side effects: none

Description: This function returns the vertical height of the specified window, in pixels.

11.8.33. window-width

Argument: a window

Returns: the width of the given window

Side effects: none

Description: This function returns the horizontal width of the specified window, in pixels.

11.8.34. window-x

Argument: window

Returns: the screen x-coordinate of the upper-left corner of the given window

Side effects: none

Description: This function returns the x-coordinates of the upper-left corner of the given window, in pixels, relative to the coordinate system of the display.

12. Tailoring the User Interface: An Example

Suppose an application-builder wanted to change the appearance of graphs so that circles were drawn around nodes, with radii proportional to the length of the print-names of the nodes. Suppose further that we wanted edges to be represented with very thick lines, (say 0.25 inches of thickness or so). And finally, when the mouse points to a node, we would like the node to be highlighted with a series of concentric circles (much like the appearance of a "bull's-eye" target). Of course, the layout of the graph should be spaced apart proportionally so as to accommodate the new over-sized circular nodes. The following definitions would accomplished these goals:

```
(defun circles-pname-length-function (node &optional window)
   (+ 20 (max (default-pname-length-function node window)
      (default-pname-height-function node window))))

(defun circles-pname-height-function (node &optional window)
   (+ 20 (max (default-pname-length-function node window)
      (default-pname-height-function node window))))
```

```
(defun circles-node-paint-function (node &optional window x-offset y-offset)
   (prog (adjusted-pname-length adjusted-pname-height)
      (setq adjusted-pname-length (- (node-pname-length node) 20))
      (setq adjusted-pname-height (- (node-pname-height node) 20))
      (setq window (or window (node-containing-window node)))
      (setq x-offset (or x-offset 0))
      (setq y-offset (or y-offset 0))
      (draw-string (node-pname node)
          adjusted-pname-length adjusted-pname-height
         (+ 10 x-offset (displaced-node-x-coordinate node))
         (+ 10 y-offset (displaced-node-y-coordinate node)
            (round (- (browser-quotient adjusted-pname-length 2)
                  (/ (default-pname-height-function node window) 2))))
         (node-font node)
          window)
      (draw-circle window
         (+ 10 x-offset (displaced-node-x-coordinate node)
            (round (browser-quotient adjusted-pname-length 2)))
         (round (+ 10 y-offset (displaced-node-y-coordinate node)
            (browser-quotient adjusted-pname-length 2)))
         (+ 9 (round (browser-quotient adjusted-pname-length 2)))
         'xor)
   (return-from circles-node-paint-function t)
   ) )

(defun circles-edge-paint-function (edge &optional window x-offset y-offset)
   (prog ()
      (setq x-offset (or x-offset 0))
      (setq y-offset (or y-offset 0))
      (loop for i from -5 to 5 do
      (draw-line (or window (edge-containing-window edge))
         (+ x-offset (displaced-edge-x1 edge))
         (+ i y-offset (displaced-edge-y1 edge))
         (+ x-offset (displaced-edge-x2 edge))
         (+ i y-offset (displaced-edge-y2 edge))))
      (return-from circles-edge-paint-function
         (draw-line (or window (edge-containing-window edge))
         (+ x-offset (displaced-edge-x1 edge))
         (+ y-offset (displaced-edge-y1 edge))
         (+ x-offset (displaced-edge-x2 edge))
         (+ y-offset (displaced-edge-y2 edge))))))

(defun circles-highlight-function (node &optional window)
   (prog (radius increment adjusted-pname-length )
      (setq adjusted-pname-length (- (node-pname-length node) 20))
      (setq increment (round (browser-quotient
                                    (node-pname-length node) 13)))
      (setq window (or window (node-containing-window node)))
      (setq radius (round (browser-quotient adjusted-pname-length 2)))
      (loop for i from 1 to (+ 10 radius) by increment do
         (draw-circle window
```

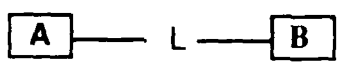
```
            (+ 10 (displaced-node-x-coordinate node)
               (round (browser-quotient adjusted-pname-length 2)))
            (round (+ 10 (displaced-node-y-coordinate node)
               (browser-quotient adjusted-pname-length 2)))
            i
            'xor))
      (return-from circles-highlight-function t)))

(defun switch-to-circle-nodes ()
   (add-node-paint-function 'circles-node-paint-function)
   (add-edge-paint-function 'circles-edge-paint-function)
   (add-pname-length-function 'circles-pname-length-function)
   (add-pname-height-function 'circles-pname-height-function)
   (add-highlight-node-function 'circles-highlight-function)
   (add-unhighlight-node-function 'circles-highlight-function)
   (setq grapher-normal-font default-bold-font)
   (layout-x-and-y)
   )
```

Now, calling the function "switch-to-circle-nodes" would instantiate all of the effects described above. In fact, the above Common-LISP definitions are included in the source code for the grapher and may be invoked from the main command menu.

13. Labeling Edges

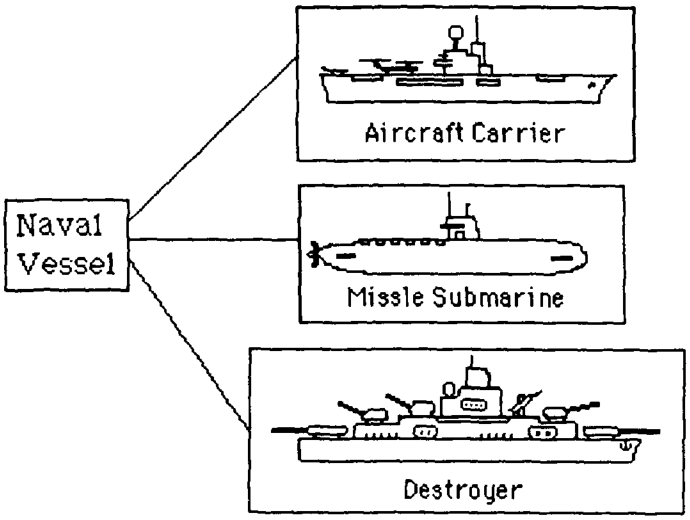
Although the ISI Grapher currently does not support edge labeling, there exists an elegant mapping [Kasper] between edge-labeled graphs and ordinary graphs. The mapping is as follows: suppose we choose to label the graph edge connecting nodes **A** and **B**, using the label **L**. We would then introduce into the graph a brand-new node in between **A** and **B** and name it **L**. That is, the subgraph {(**A**,**B**)} would be modified to become {(**A**,**L**), (**L**,**B**)}. On the display, the new node **L** may be represented using a different font or style, so as to distinguish it from an ordinary node. This may be accomplished via the introduction of a new node-paint function, as described elsewhere in this manual. Graphically, the situation may now look as in the following diagram:



The source of this transformation would be the sons-function supplied by the application-builder, while the distinction between real nodes and edge-simulating nodes may be encoded into the node-pnames via the corresponding function. The only undesirable side-effect would be that the normal layout algorithm will not necessarily place **L** on the midpoint between **A** and **B**. This limitation may be overcome if the application-builder's node-paint function, when called to draw **L**, ignored **L**'s coordinates and instead placed **L** midway between A and B's positions, using some special notation to distinguish **L** as an edge-label.

14. Icons

There are numerous ways to make ISI Grapher displays even more visually striking. For example, the user could utilize icons to display nodes, whereupon the BBN Naval Model could take on the style of the following diagram:

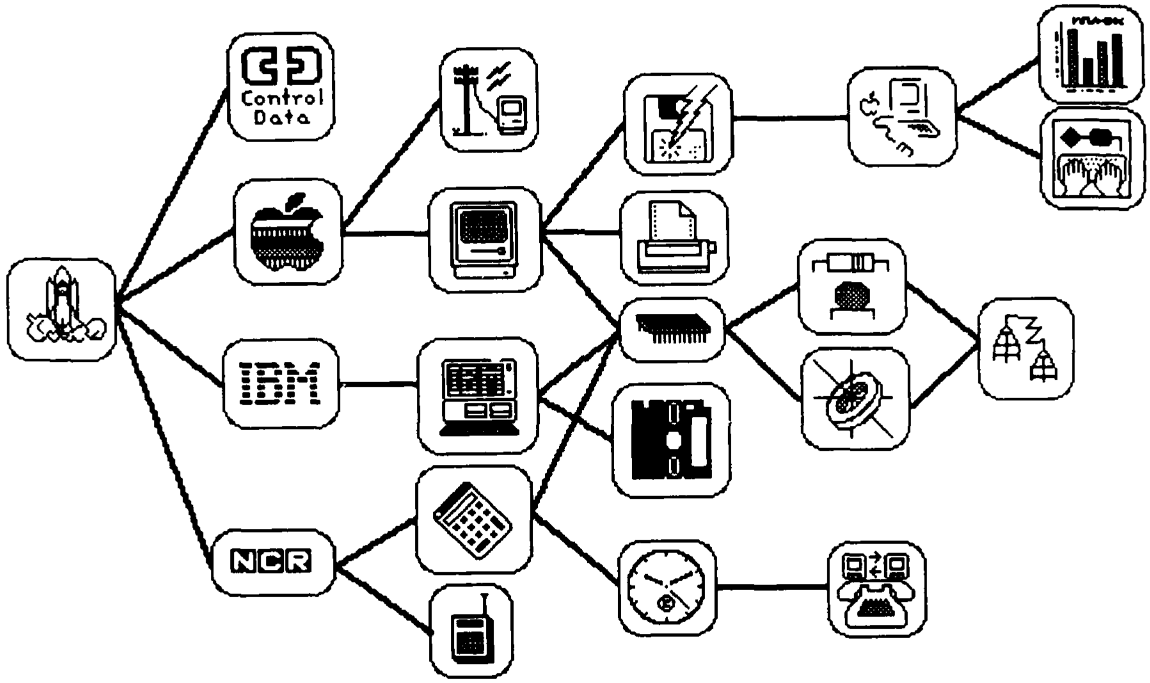


An example of an icon-based ISI Grapher display

This may be accomplished by using the font-editor to create a specialized font which would include the above icons as special characters. As the ISI Grapher is capable of working with arbitrary fonts, the above display would readily result after the addition of the proper (trivial) node-paint function.

The fundamental motivation for making such extensions to the ISI Grapher is the observation that, all other things being equal, the system with the most friendly user interface is often also the most useful and impressive.

Another icon-based graph example might be as follows:



Another example of an icon-based ISI Grapher display.

Note that no semantics are attached to the diagram above; it is simply included here as a fancy example of specializing the ISI Grapher during application-building.

15. Hardcopying

As hardcopying is system and device-dependent, the grapher has no general hard-copying capabilities. However, it does provide for a mechanism which automatically scrolls the current grapher window incrementally in the x and y directions and calls a user-specified function which is responsible for the actual hard-copying of that portion of the graph which is currently visible in the grapher window. The idea here is that since most hardcopying devices are capable of producing an image of only a small (page-sized) bitmap, in order to obtain a hardcopy of a large graph (say, 50 square feet in area), a user must hardcopy pieces of it in small sections. Then the user must cut-and-paste the resulting "jigsaw-puzzle" together to obtain the final wall-sized diagram. The automatic scrolling also provides a small overlap margin between adjacent panes which has proved to be quite handy during the final cutting-and-pasting process. In summary, the ISI Grapher does provide an automatic means of scrolling in order to hardcopy a graph in small sections, but the user is responsible for providing a hardcopying function that can handle each section.

It is suggested that a plotter may be quite useful as an output medium, but unless the maximum sheet-size that the plotter can accommodate is considerably large (say, at least an 11" by 17" sheet), plotting may not be any faster or more convenient than the printing-and-pasting

process just described. In addition, there is the problem of mapping arbitrary bit-maps into pen-plot figures, which is a non-trivial problem. Although a color-coded plot of a large graph may be particularly attractive, the ISI Grapher currently does not support such a plotting facility; nevertheless, users are encouraged to experiment with this idea, and the author will be very interested to hear of any related results.

16. Obtaining the sources

The ISI Grapher currently runs on several different kinds of workstations, including Symbolics, TI Explorers, SUNs, and the MacIntosh II. The source code for the ISI Grapher may be obtained by contacting the author: Gabriel Robins, Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, Ca, 90292-6695, U.S.A.; ARPAnet address: "gabriel@vaxa.isi.edu". The author has already received and responded to several hundreds of requests for the source code from various corporations and universities worldwide. To obtain the MacIntosh or SUN implementation (among others), contact ExperTelligence Inc., 559 San Ysidro Road, Santa Barbara, Ca 93108, (805) 969-7874.

17. Glossary

Apple MacIntosh - the personal computer used to generate this document. The exact configuration used was a MacIntosh-Plus with 2 megabytes of memory and a 20 megabytes hard disk. The software used to generate this document is the MicroSoft Word 3.01 word processing program. The ISI Grapher is currently being marketed for the Mac+ and Mac II through ExperTelligence Inc.

application-builder - anyone who uses the ISI Grapher as a basis to producing another piece of software.

edge-record - a Common LISP record that corresponds to a directed graph edge in a particular grapher window. The fields contained in each edge record include the node the edge emanates from and the node the given edge terminates upon.

ExperTelligence Inc. - A private company currently marketing the ISI Grapher for several brands of computers, including the Apple MacIntosh family, as well as for SUNs. Their address is 559 San Ysidro Road, Santa Barbara, Ca 93108, (805) 969-7874.

Gabriel Robins - the author of the ISI Grapher, with ARPAnet address of gabriel@vaxa.isi.edu.

graph-lattice - the main high-level function through which the ISI Grapher is invoked.

Information Sciences Institute (ISI) - a non-profit research organization affiliated with the University of Southern California. ISI is a major DARPA contractor. ISI's address is 4676 Admiralty Way, Marina Del Rey, CA 90292-6695, (213) 822-1511.

io-stream - an optional argument to **graph-lattice**; this is the window/stream that will be used by the ISI Grapher to print messages to the user, and also to read character input typed by the user. If this argument is omitted, a reasonable default window will be used.

ISI Grapher - an extendible and portable system for the layout and display of arbitrary graphs, developed by the Intelligent Systems Division at ISI. The fundamental motivation that gave birth to the ISI Grapher is the observation that graphs are very basic and common structures, and the belief that the ability to quickly display, manipulate, and browse through graphs may greatly enhance the productivity of a researcher, both quantitatively and qualitatively. The ISI Grapher is implemented in Common LISP.

ISI NIKL Browser - a browser for NIKL, built on top of the ISI Grapher. The ISI NIKL Browser allows a user to quickly layout, display, and browse through NIKL taxonomies.

layout algorithm - the scheme used to map an abstract graph nodes and edges into physical positions on the display.

layout-flag - an important (optional) argument to **graph-lattice**; **layout-flag** may be either '**tree** or '**lattice**. '**tree** means the graph will be displayed as a pure tree, regardless of its structure (in case there are directed or undirected cycles, they will be "broken" for displaying

purposes by the introduction of "stub" nodes. **'lattice** means graph as-is, with nodes with multiple parents displayed as such via shifting of their position. The omission of this argument will cause a default to **'tree** to occur, as will the existence of directed cycles in the graph.

NIKL - A knowledge-representation language developed jointly by ISI and BBN. NIKL is a direct descendent of KL-ONE, and is the brainchild of Ron Brachman.

node-record - a Common LISP record that corresponds to a graph node in a particular grapher window. The fields contained in each node record include the object the node represents, the print-name of the object and its dimensions, the associated font, the location on the screen of the node, as well as the children and parents of this node.

options (or root-list) - a mandatory argument to **graph-lattice**which may be a root object or an options list. If this argument is not a list, it is interpreted as the object corresponding to the root of the graph. If this argument is a list, it is interpreted as a command/options list.

the-children-function - this mandatory argument to **graph-lattice** is interpreted as the name of a (presumably existing) function which, when called with a single argument corresponding to a graph-object, returns the list of objects (of the same type) corresponding to the children of the argument node in the graph. That is, if the 'sons is passed to **graph-lattice** as **the-children-function**, then anytime later, it is assumed that the call "(sons x)" returns a list "(y_1 y_2 ... y_n)" if and only if (x, y_i) is an edge in the graph, for all $1 \le i \le n$. This is the primary mechanism used by the ISI Grapher to determine the structure of the graph.

tracking-mouse - An alternate way of exiting the grapher is to set the global variable **tracking-mouse** to NIL. This is useful when an application running concurrently with the grapher wishes the grapher to exit out of the main mouse-tracking loop.

user - anyone using the ISI Grapher or an application which is built on top of the ISI Grapher.

window-record - A Common LISP record that corresponds to a grapher window. The fields contained in each window record include a list of records corresponding to the roots of the graph, the name of the children-function, a list of node- and edge-records associated with the given graph, a list of available fonts, various size parameters for the graph and its window, and a list corresponding to the subset of the nodes and edges that are currently visible (or partially visible) in the graph window. Thus, each window-record contains a copy of each global variable associated with a single graph, its layout, and its window.

18. Acknowledgements

The author is grateful to **Ron Ohlander** for his excellent leadership, and for providing interproject support for further development effort. Without Ron's support and encouragement the ISI Grapher would not have evolved thus.

Bob MacGregor deserves credit for several suggestions, and also for being the first one to suggest that the ISI Grapher merits interproject support

The author is indebted to **Larry Miller**, for supervising the continued development of the ISI Grapher as an inter-project project. Without Larry's patience and support, the ISI Grapher would have been doomed to remain a buggy prototype. I am indebted to **Jouko Sepannen** for inviting me to speak at Symboliikka '87, Helsinki, Finland.

The help of the following individuals is acknowledged: **Bob Kasper**, for suggesting the clever mapping between plain graphs and edge-labeled graphs, and also for proofreading, **Ray Bates**, for technical help with LISP on numerous occasions, **Norm Sondheimer**, upon whose suggestion the BBN Naval Model turned into a beautiful wall-chart, and **Tom Galloway**, whose persistent bug reports lead to subsequent improvements, and who encouraged the addition of numerous useful functions to the ISI Grapher.

Neil Goldman has done an excellent proofreading job; he has made numerous insightful suggestions, some of which are unfortunately still not implemented. **Victor Brown** has patiently corrected my grammar, punctuation, and style; he has my sincere thanks. Further credit for proofreading and comments goes to: **Tom Galloway**, **Steve Smoliar**, **Robert Albano**, **Eli Messinger**, **Ching Tsun Chou**, and **Ann Bettencourt**.

The patient help of **Leslie Ladd** and **Larry Friedman** with tedious photocopying, cutting, binding, and pasting is greatly appreciated. Larry also deserves credit for typesetting and formatting many of the diagrams that appear in this manual. **Diane Hatch-Avis** deserves credit for her help and encouragement throughout the publication process.

I would like to thank **Dennison Bollay**, **John Forge**, and **Dean Ritz** of ExperTelligence Inc., for finding the initiative to port the ISI Grapher to the MacIntosh and undertaking to market the resulting product. It is due to their energy and efforts that the ISI Grapher is now commercially available to the public.

I thank **Doug Johnson** for porting the ISI Grapher to the MacIntosh under Coral Allegro Common LISP, and to **James Laurus** for porting the ISI Grapher to SUNs under Franz and X.

Finally, many thanks go to **Tom Kaczmarek**, who suggested the usefulness of a portable grapher in the first place, and under whose guidance the ISI Grapher was initially born.

19. Bibliography

Kasper, B., personal communication, Information Sciences Institute.

Robins, G., The ISI Grapher: a Portable Tool for Displaying Graphs Pictorially, ISI/RS-87-196, USC/Information Sciences Institute, reprinted from the Proceedings of Symboliikka '87, Helsinki, August 17-18, 1987.

Supowit, K., and Reingold, E., The Complexity of Drawing Trees Nicely, Acta Informatica, **18**, 1983, pp. 377-392.

Wetherell, C., and Shannon, A., Tidy Drawing of Trees, IEEE Transaction on Software Engineering, **5**, September 1979, pp. 514-520.

20. Appendix

This section contains various screen snapshots of the ISI Grapher during execution on a MacIntosh II. Currently, the ISI Grapher is also running on Symbolics, TI Explorer, and SUN workstations, with ports to HP Bobcats and other machines soon to follow.

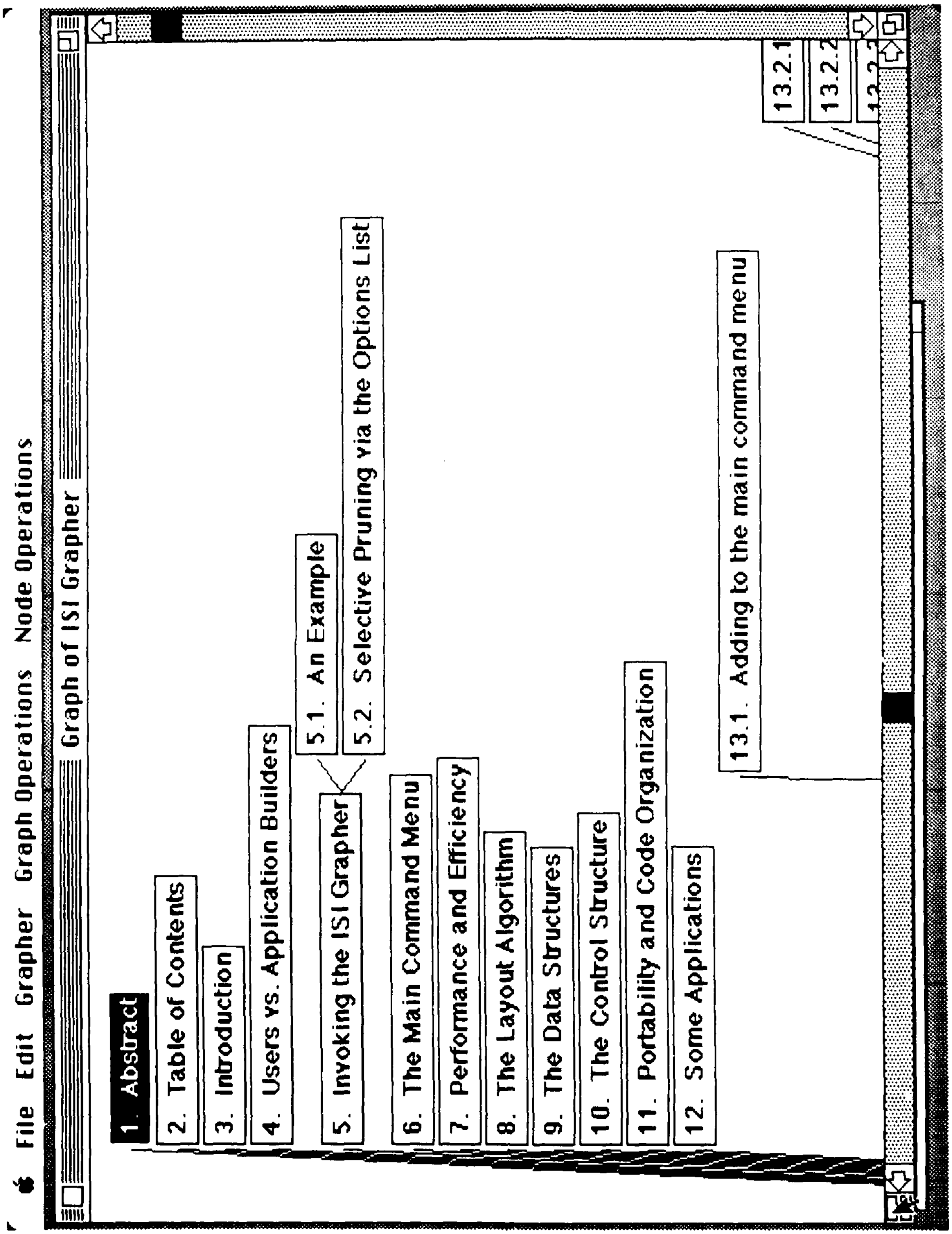


Figure 1: A typical ISI Grapher display window (called the graph-window); this graph depicts the structure of this manual, arranged hierarchically according to sections and subsections. Nodes are represented by boxes enclosing the text strings corresponding to the section titles.

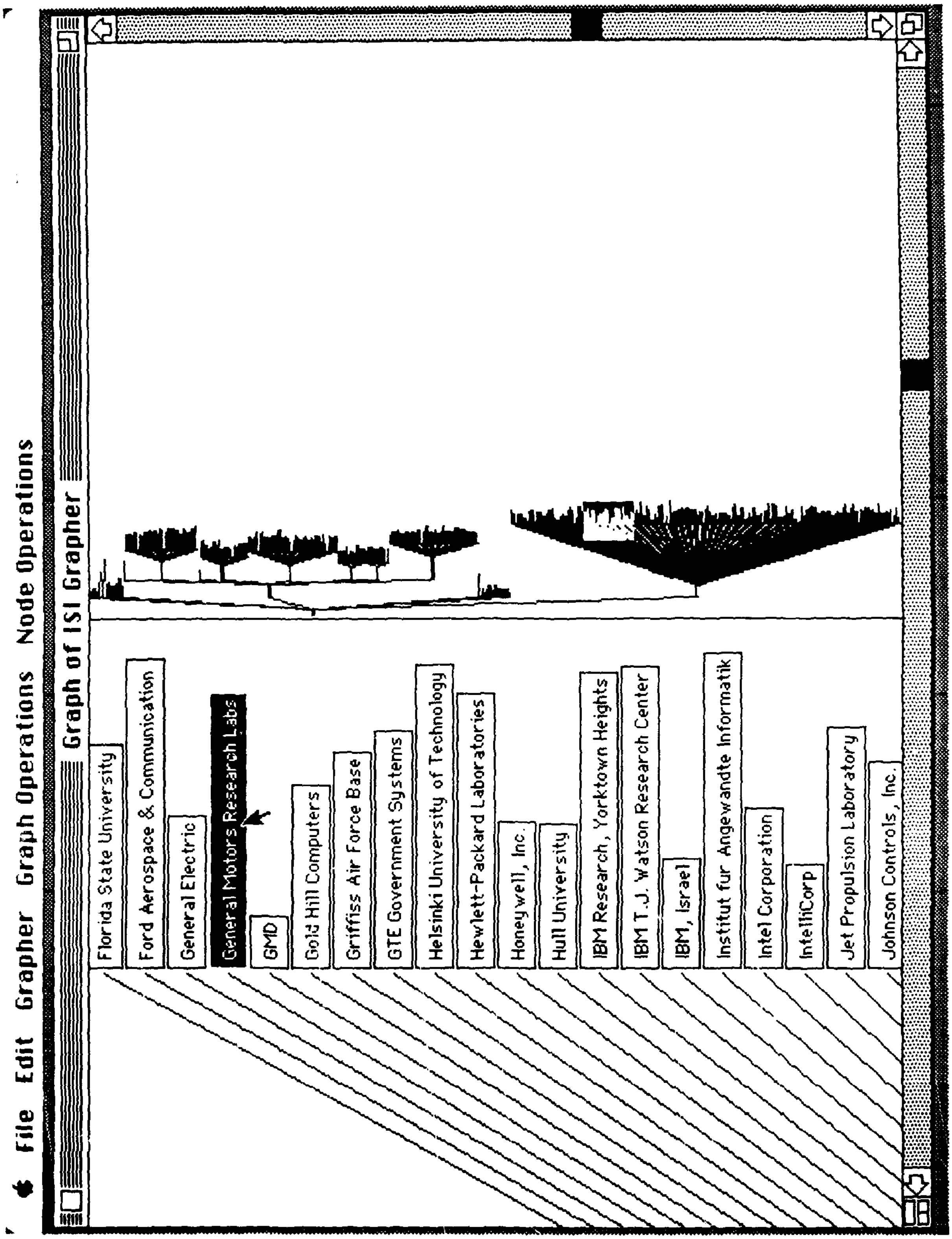


Figure 2: Here a second window is visible as well showing the global overview (or map) of the graph. The small box/zoom-rectangle highlighted in the map window is that area of the graph which appears in greater detail in the graph-window to the left. This graph depicts some of the companies that were interested in the ISI Grapher.

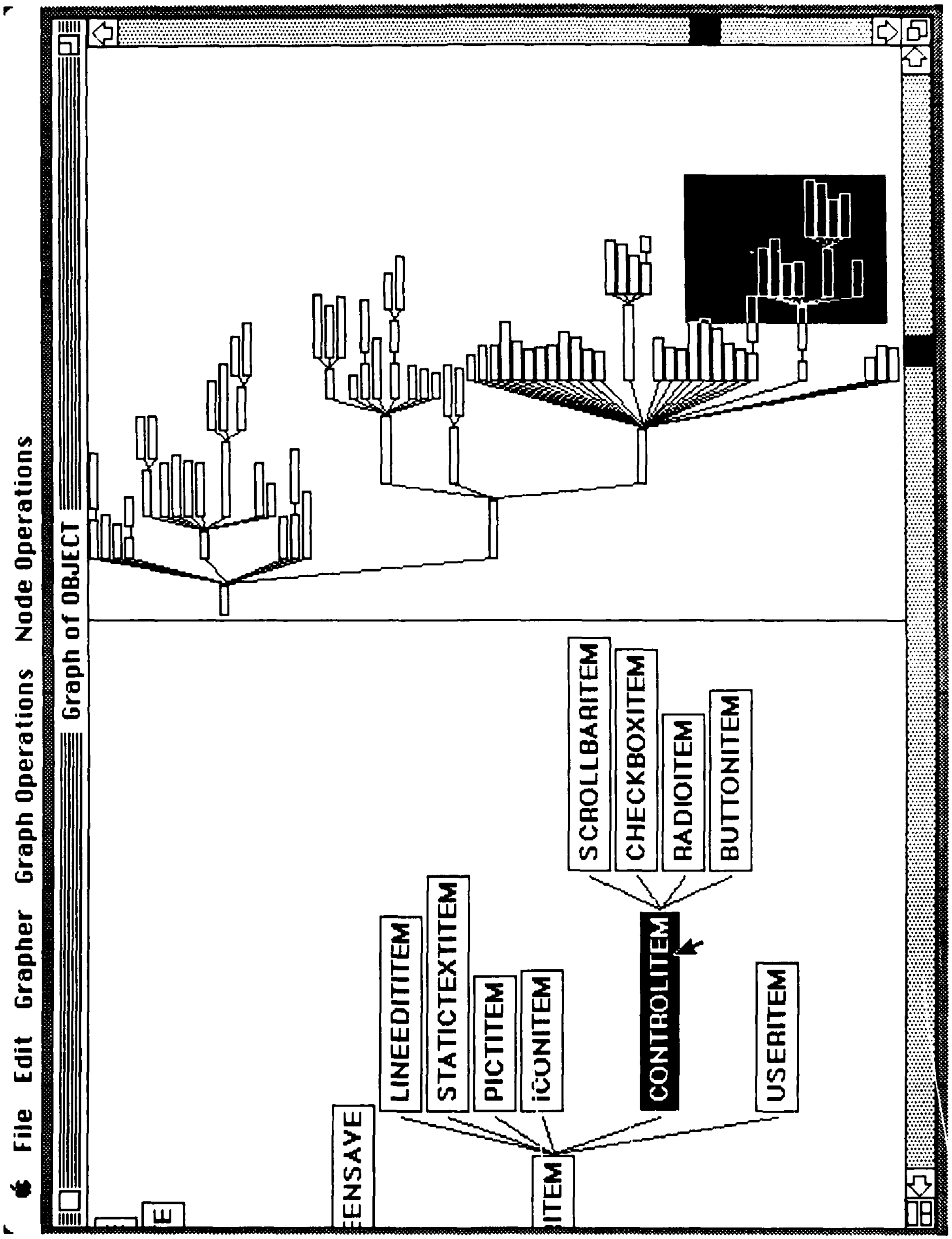


Figure 3: Nodes can be selected/highlighted and then be used in conjunction with various operations. Scrolling through the graph is accomplished by dragging around the zoom-rectangle in the map window; the graph-window is then updated accordingly.

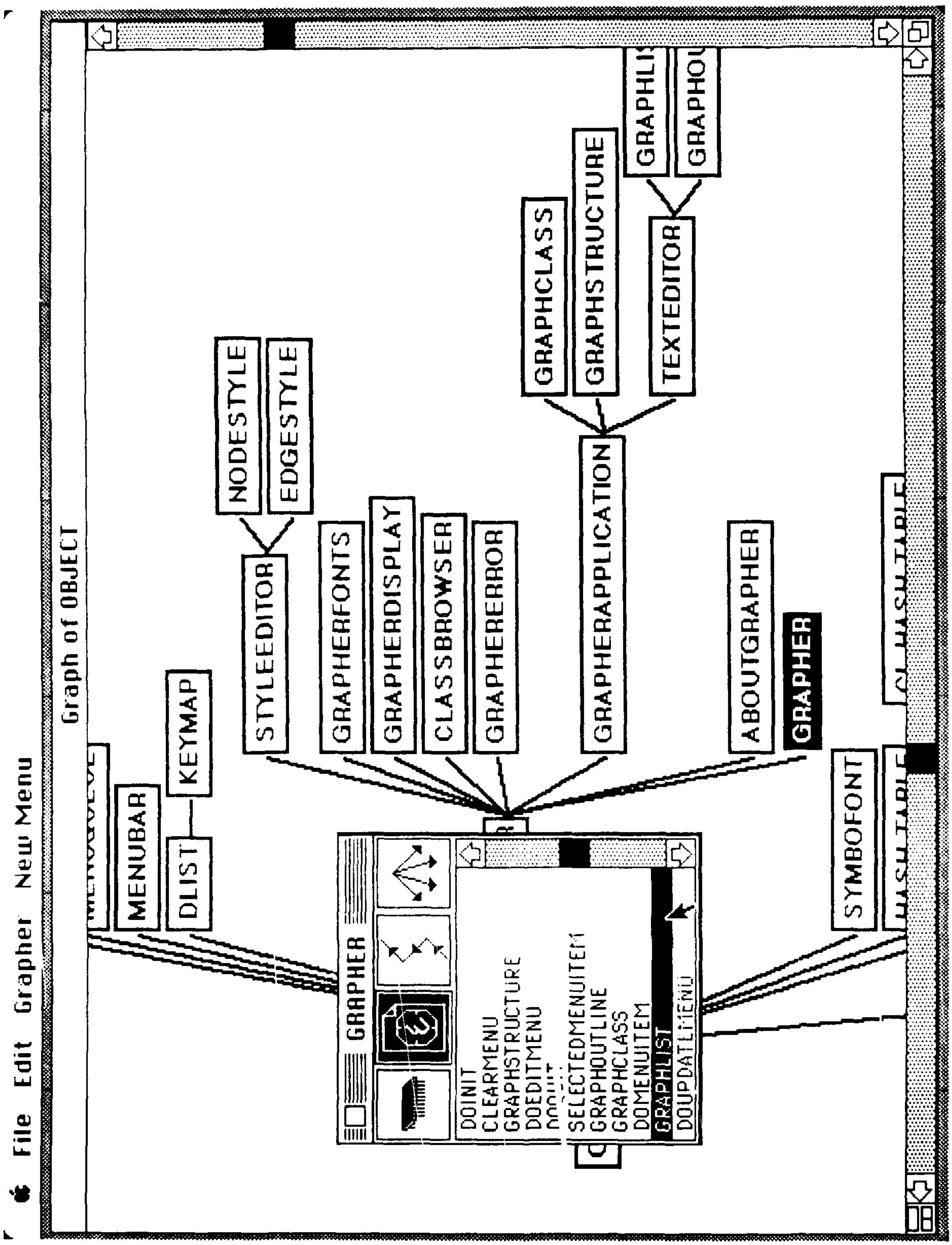


Figure 4: Operations that are available by clicking on nodes include "going" to one of the highlighted node's parents or children in the graph, as well as arbitrary user specified operations.

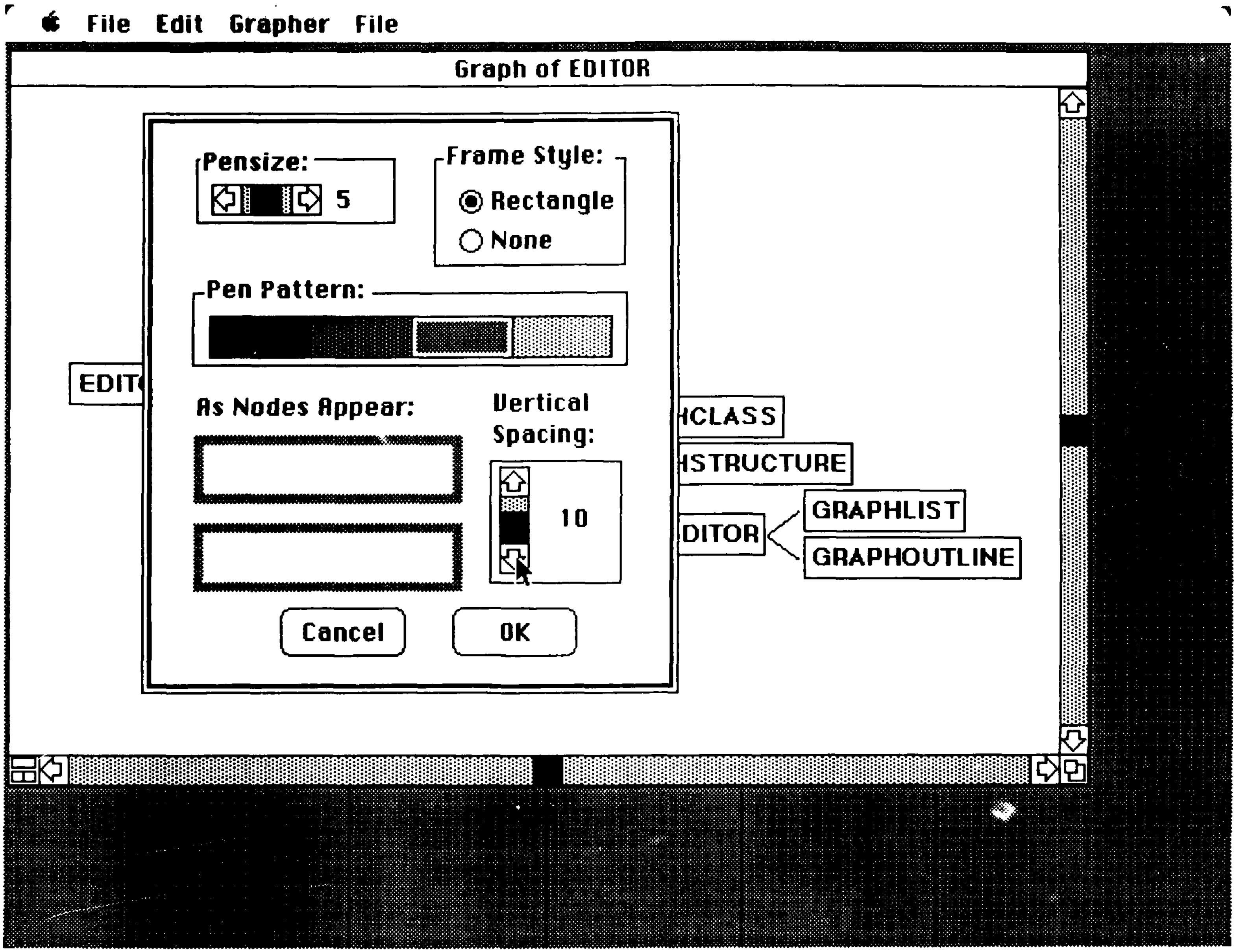


Figure 5: The way nodes are displayed is extremely flexible. While some of the node-display parameters can be specified from the user-interface as shown here, the user may in fact provide an arbitrary node-paint function; on systems where color monitors are available, having this function draw certain nodes in different colors is likely to enhance both readability and visual effect.

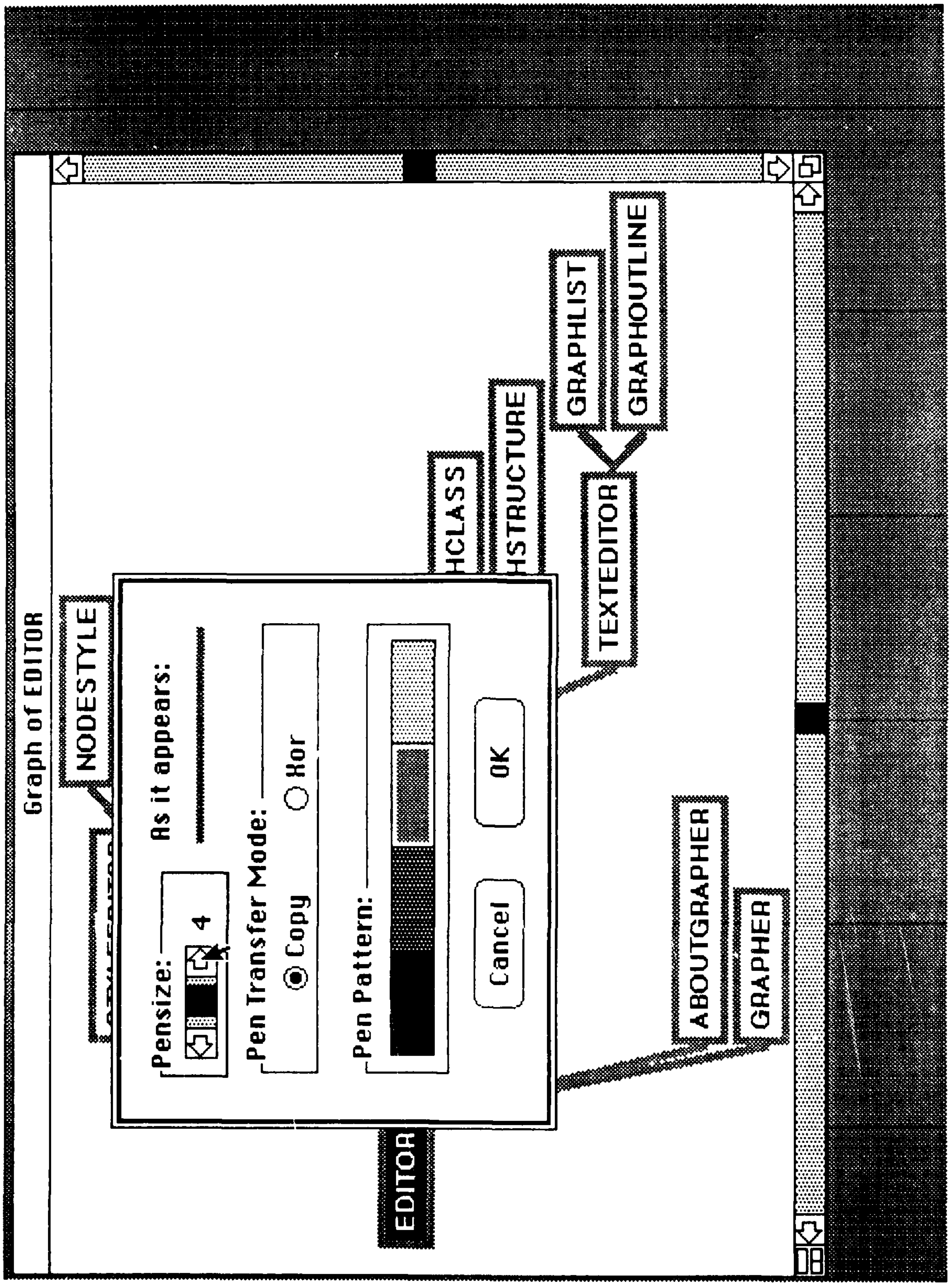


Figure 6: Some edge-display parameters can also be specified from the user-interface, as depicted in this diagram; as in the case of nodes, the user may also specify an arbitrary edge-paint function.

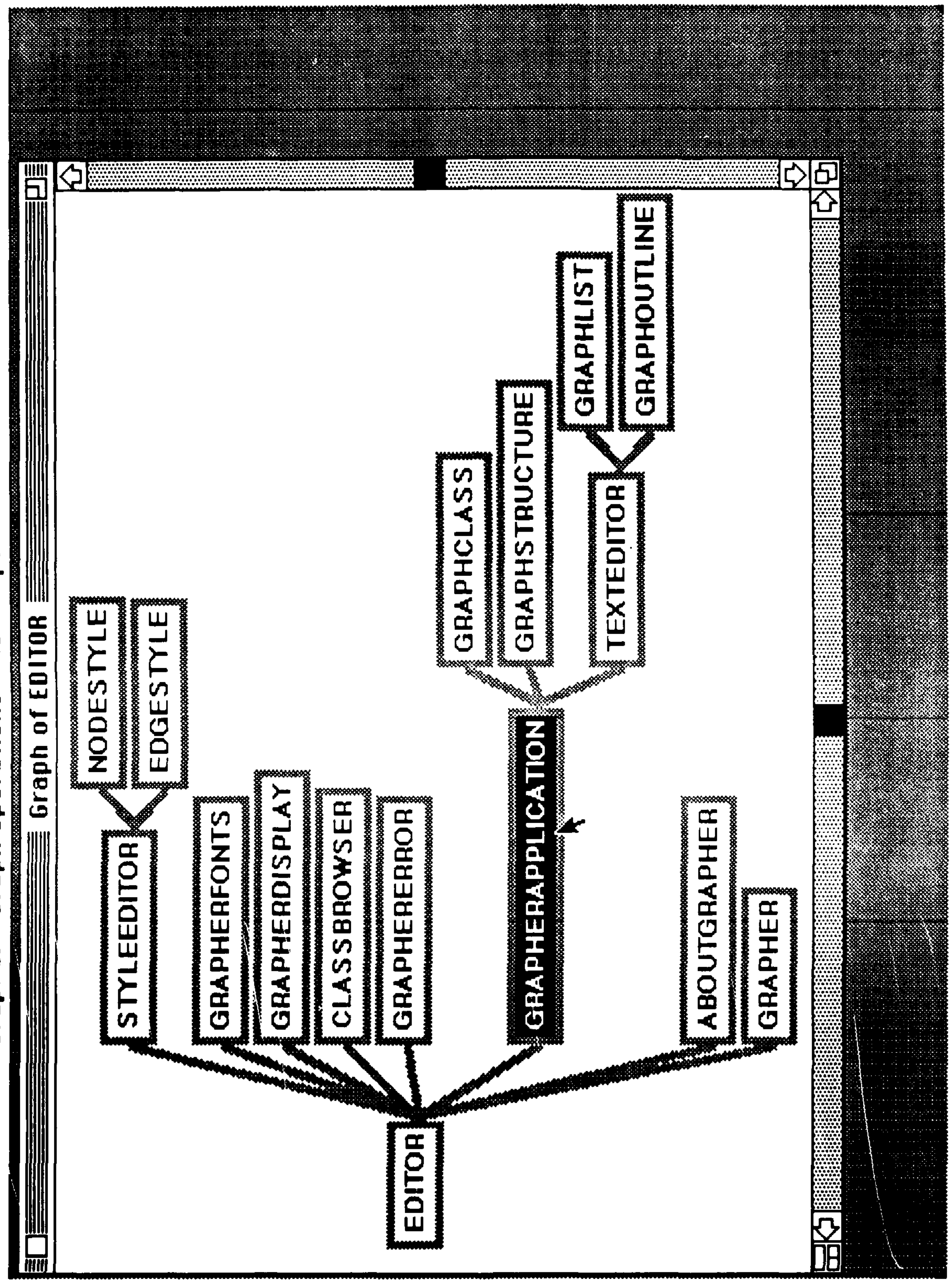


Figure 7: This is how the graph appears after both the node and edge -painting parameters were modified; nodes are now boxed with heavy grey boxes and the edges are drawn as thick grey lines.

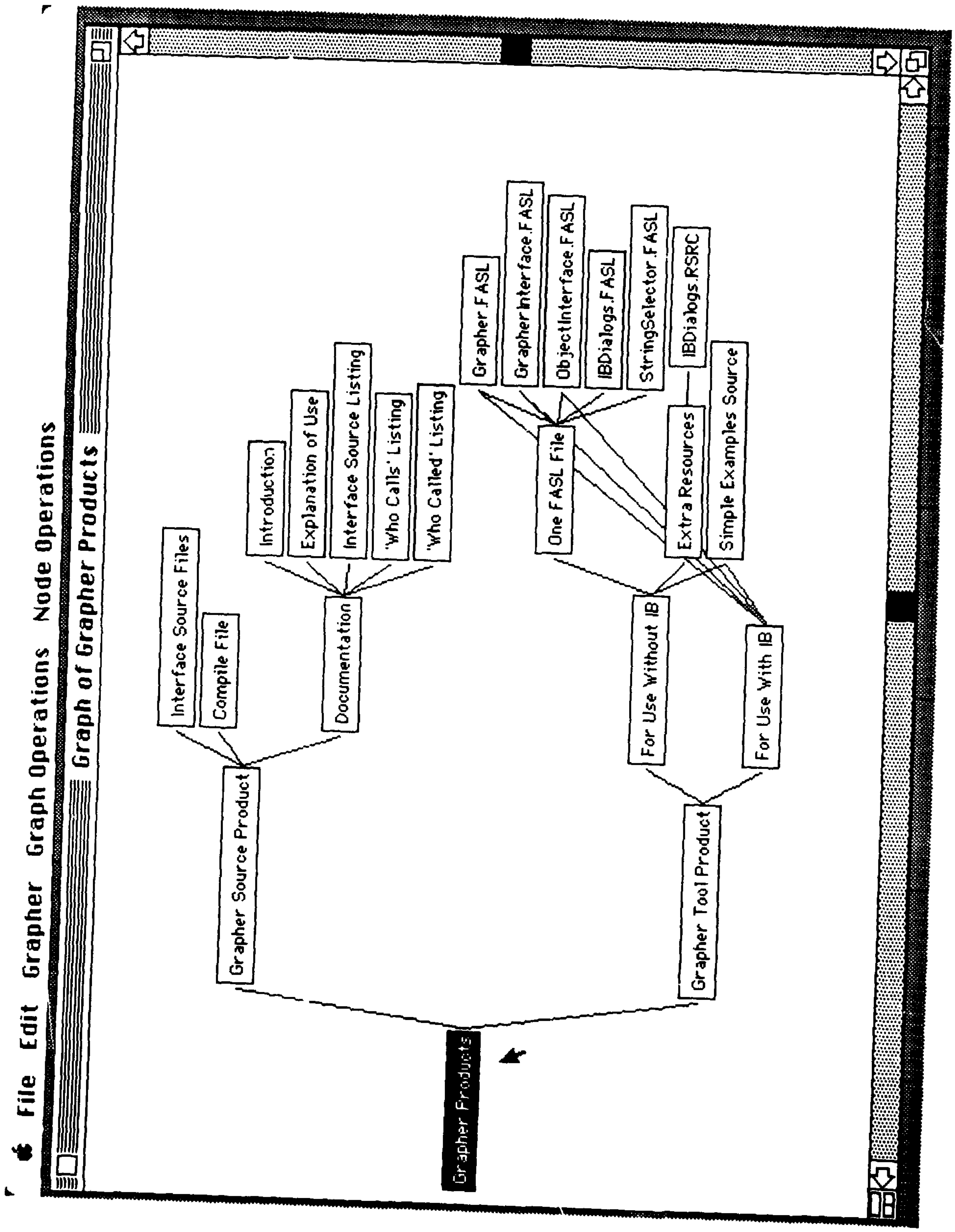


Figure 8: This graph depicts the details of some ISI Grapher-related products available from ExperTelligence Inc.; note that the "lattice" option was used, so some nodes have *multiple parents* in the diagram.

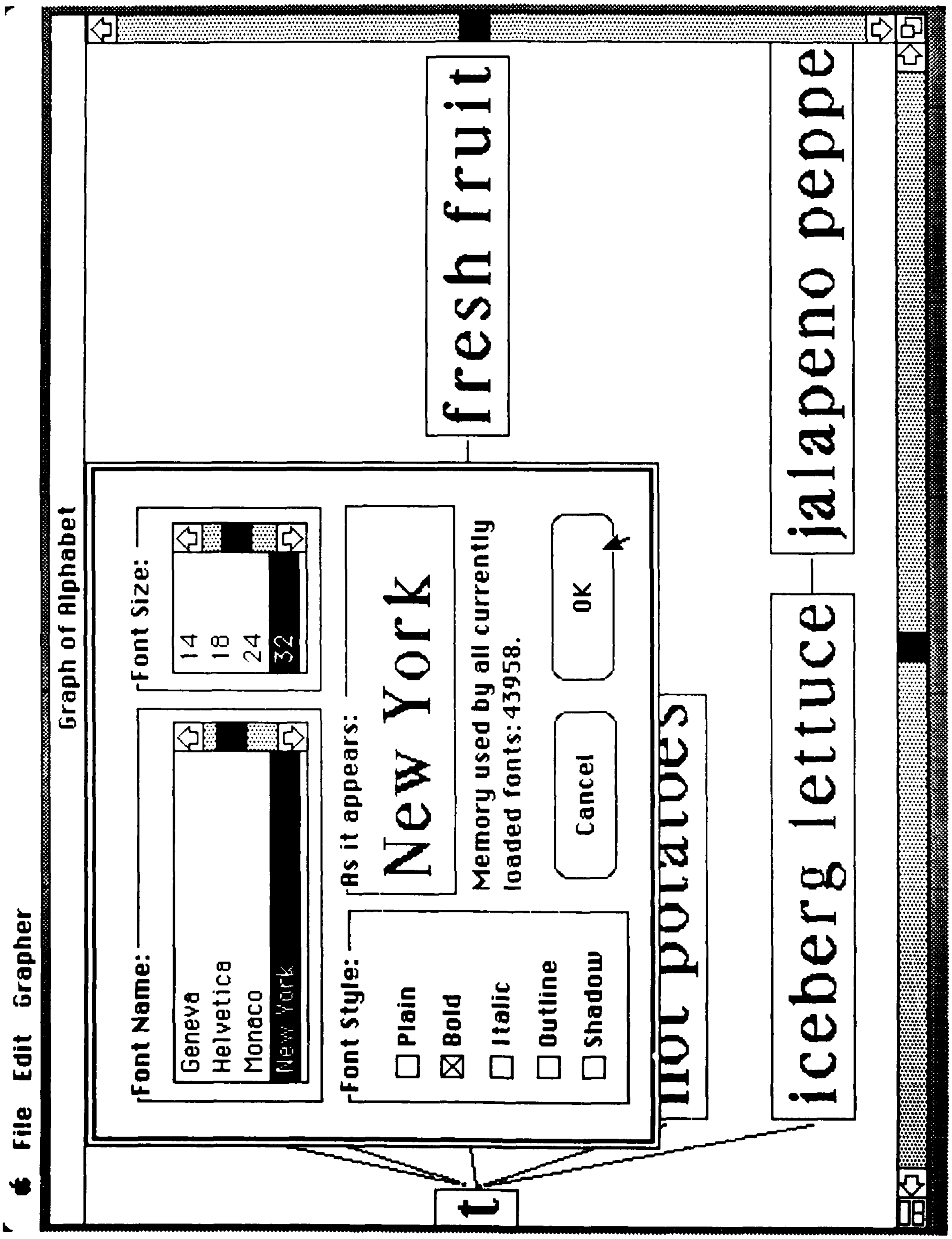


Figure 9: The ISI Grapher can handle arbitrary fonts and the layout changes accordingly automatically. In this example a larger font was selected to be used with all nodes of this graph. Individual nodes may have different fonts.

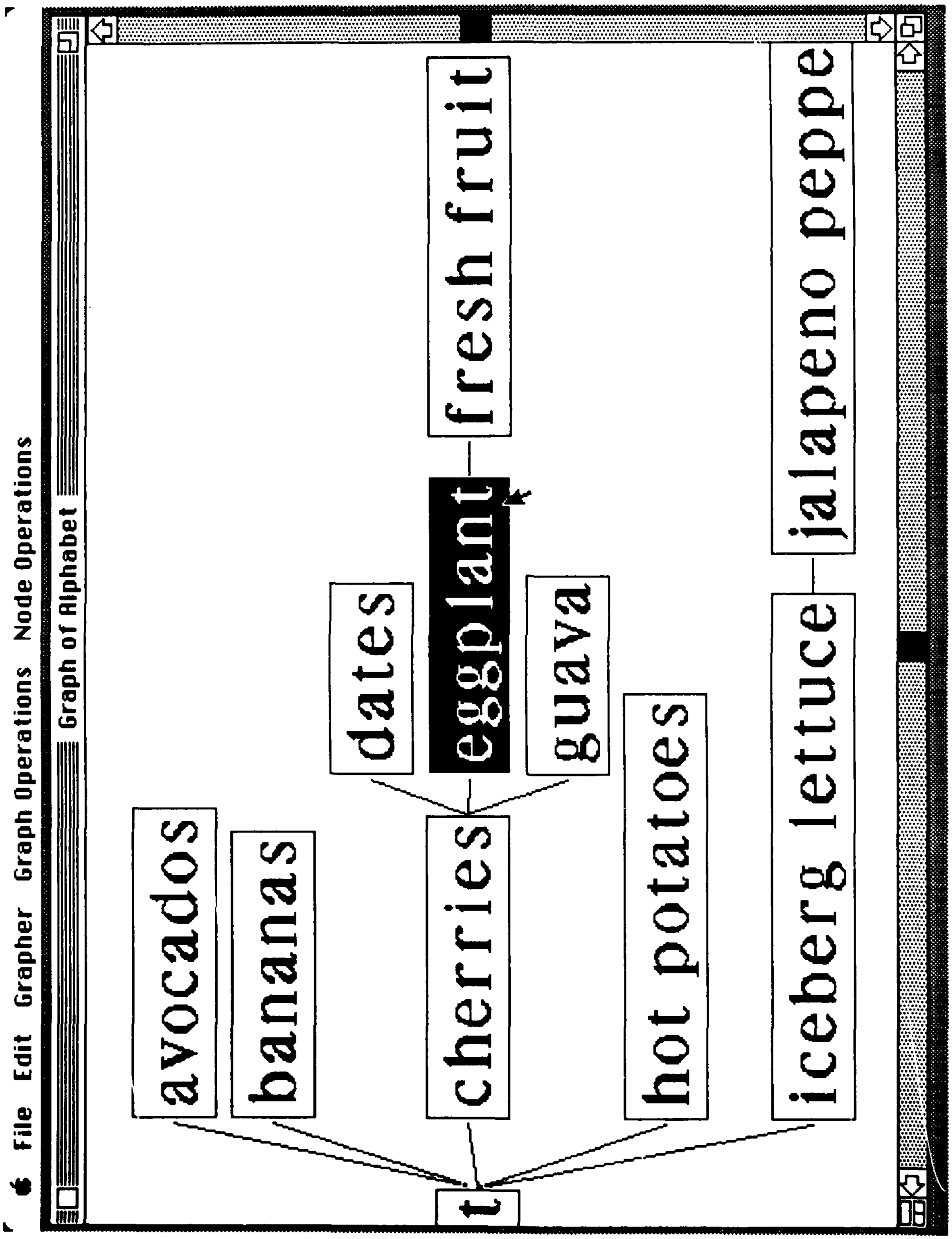


Figure 10: This is how the graph appears after the font was changed. Such a large font may be quite useful in demos to small groups.

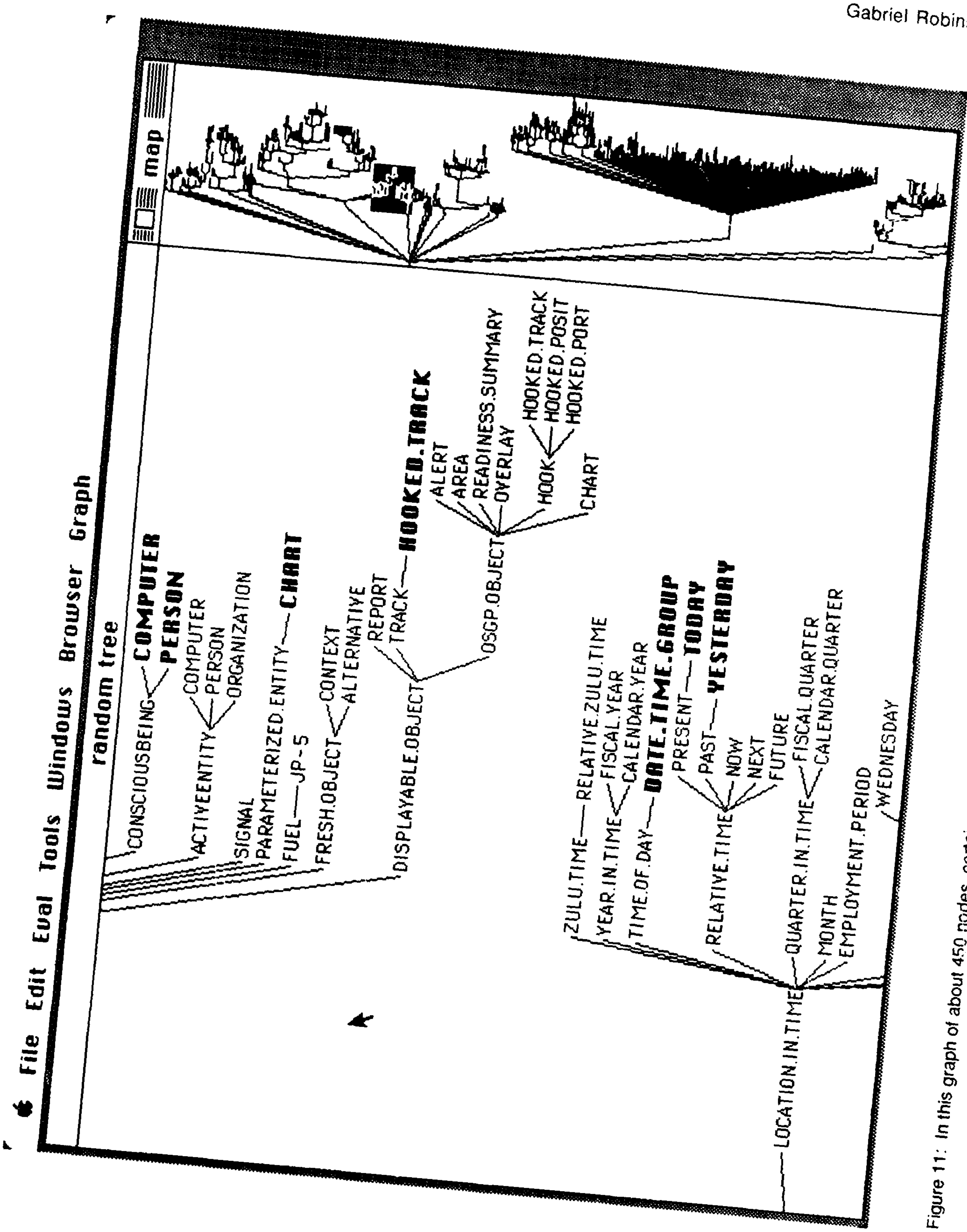


Figure 11: In this graph of about 450 nodes, certain nodes are depicted using a different font.

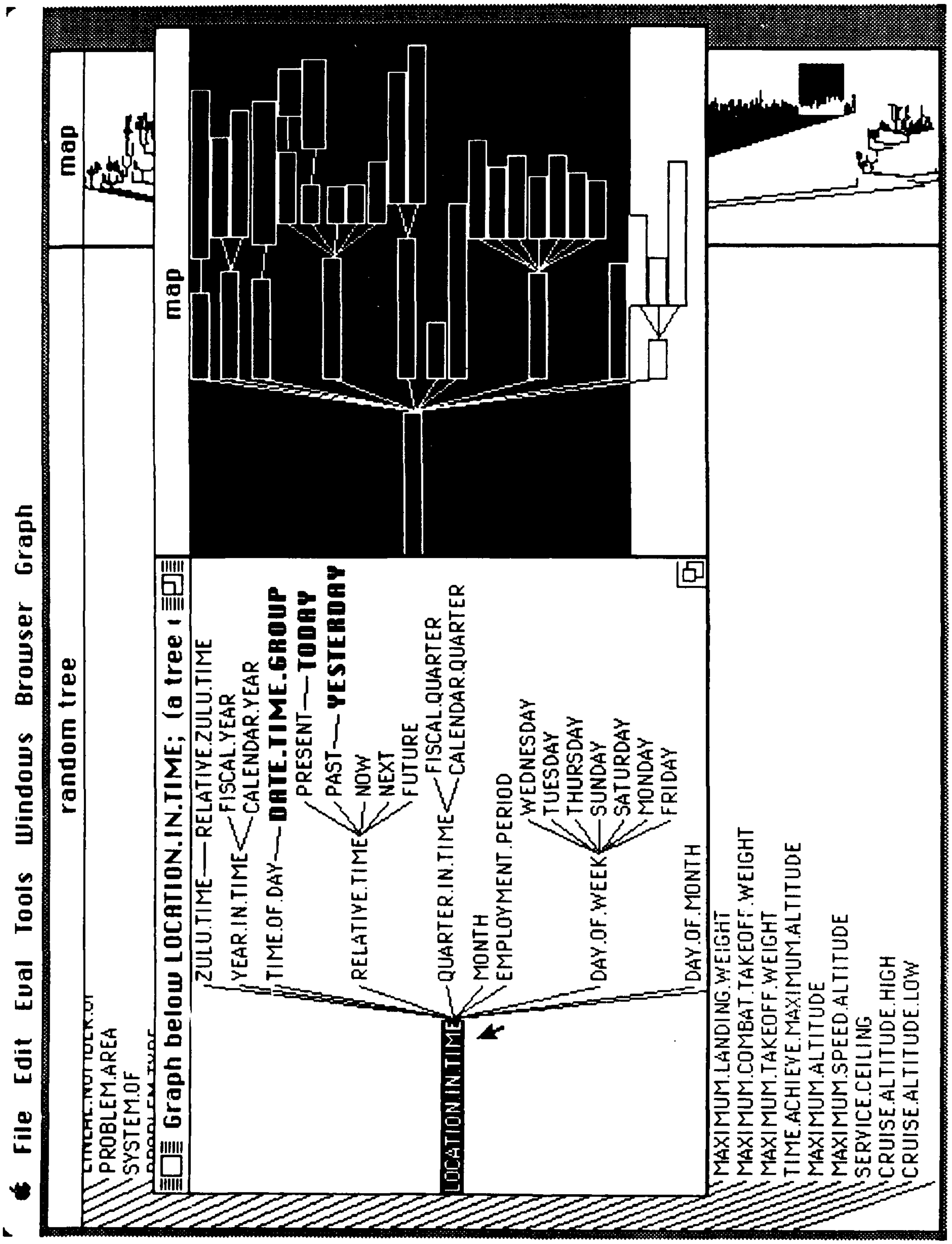


Figure 12: Multiple graphs can be handled simultaneously by the ISI Grapher, each having its own windows and display parameters.

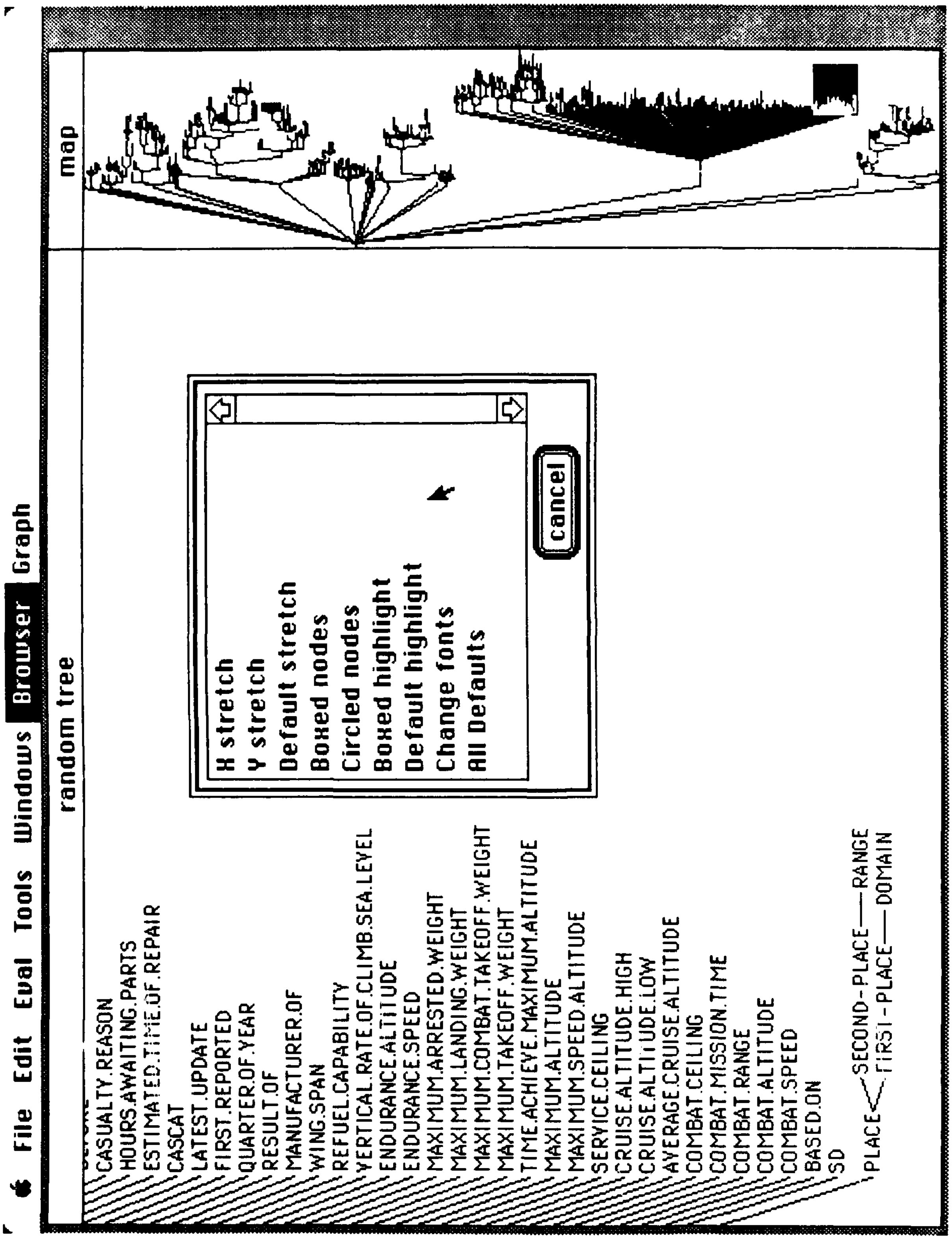


Figure 13: Several ways of modifying the appearance of the layout are available from the menus, while others are available only via special function-calls. Moreover, the user can add his own arbitrary mechanisms to these menus in order to create customized applications.

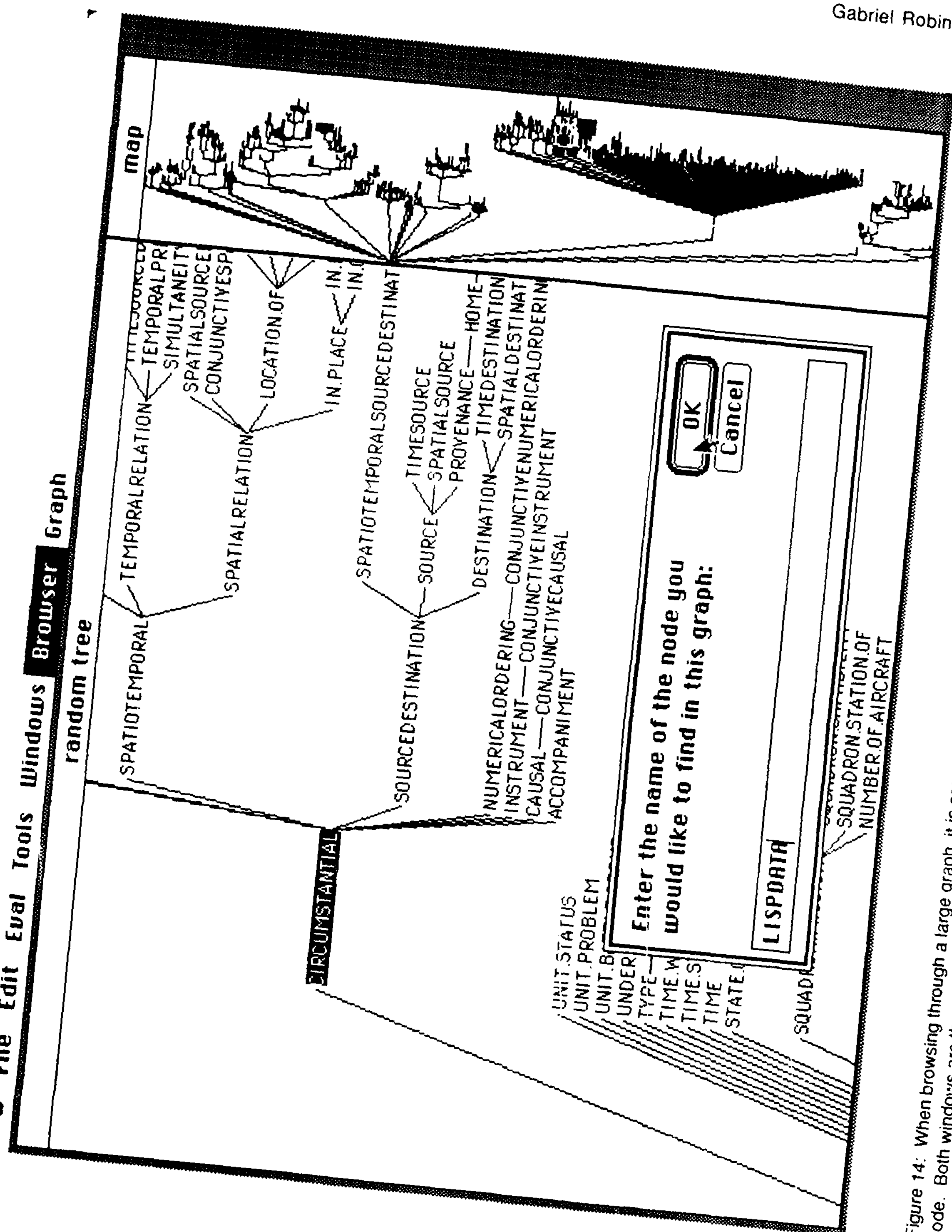
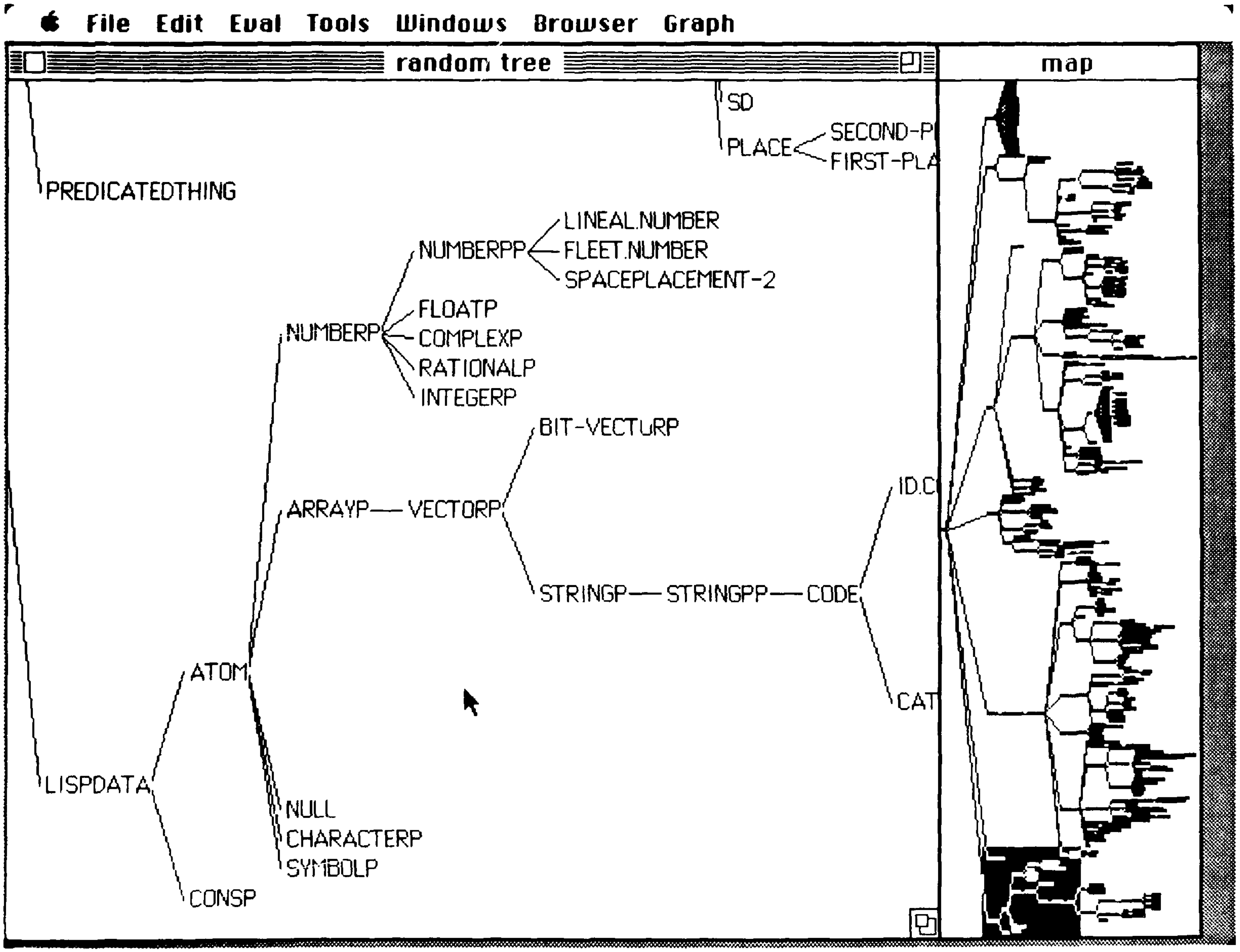


Figure 14: When browsing through a large graph, it is convenient to use the search facility in order to directly go to a specified node. Both windows are then updated and refreshed accordingly.

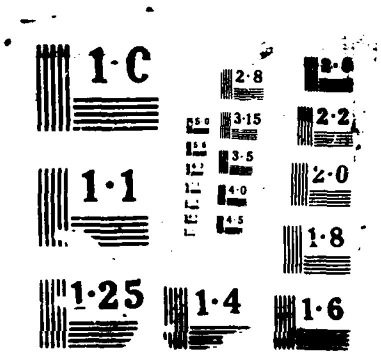


After the user specified that the node "LISPDATA" should be brought into focus, the display is centered around this node; in the neighborhood we observe a hierarchy of Common LISP data types.

AD-A192 198 THE ISI (INFORMATION SCIENCES INSTITUTE) GRAPHER MANUAL 2/2
(U) UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY
INFORMATION SCIENCES INST G ROBINS FEB 88

UNCLASSIFIED ISI/TM-88-197 MDA903-81-C-0335 F/G 12/5 NL

END
DATE
FILMED
5 88


1·0
2·8
2·5
3·15
2·2
3·5
2·0
4·0
1·1
4·5
1·8
1·25
1·4
1·6

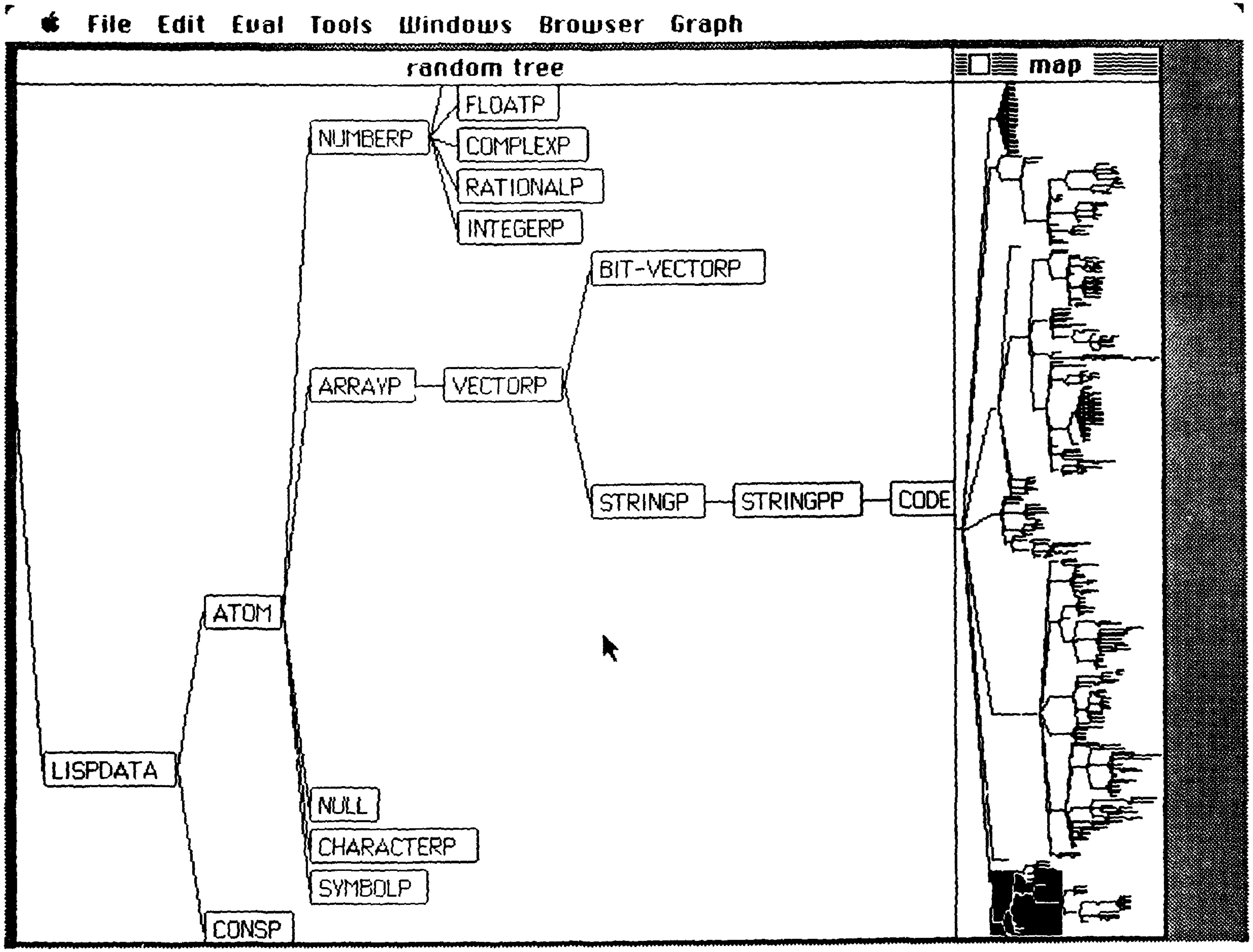


Figure 16: This is the result when the user specified that boxes be drawn around nodes.

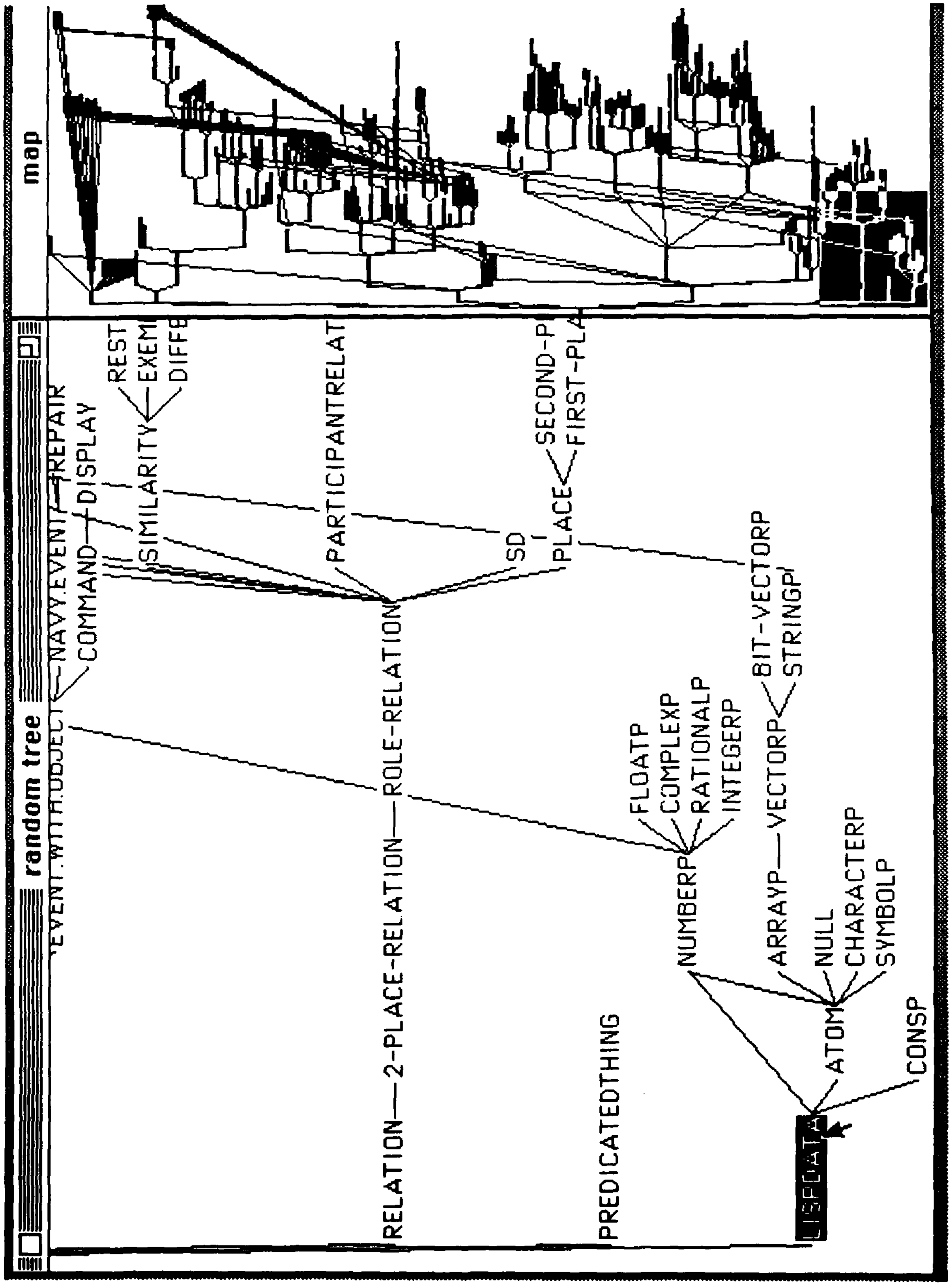


Figure 17: This is the same graph as in the previous figure except that the "lattice" option has been specified, so that some nodes are depicted as having multiple parents.

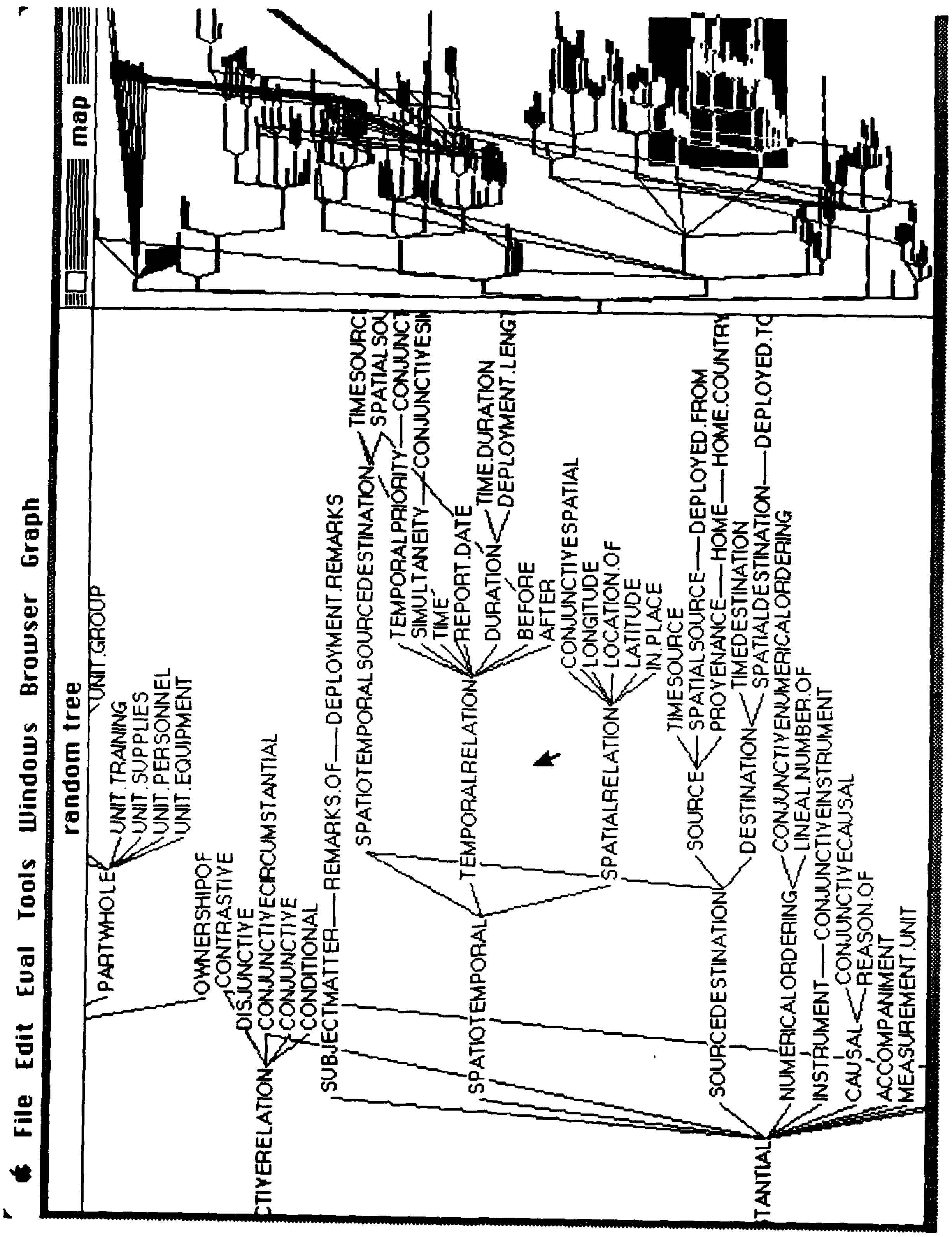


Figure 18: The user has scrolled to another area of the graph of the previous figure. Scrolling is accomplished by simply dragging the "zoom-box" around in the map-window.

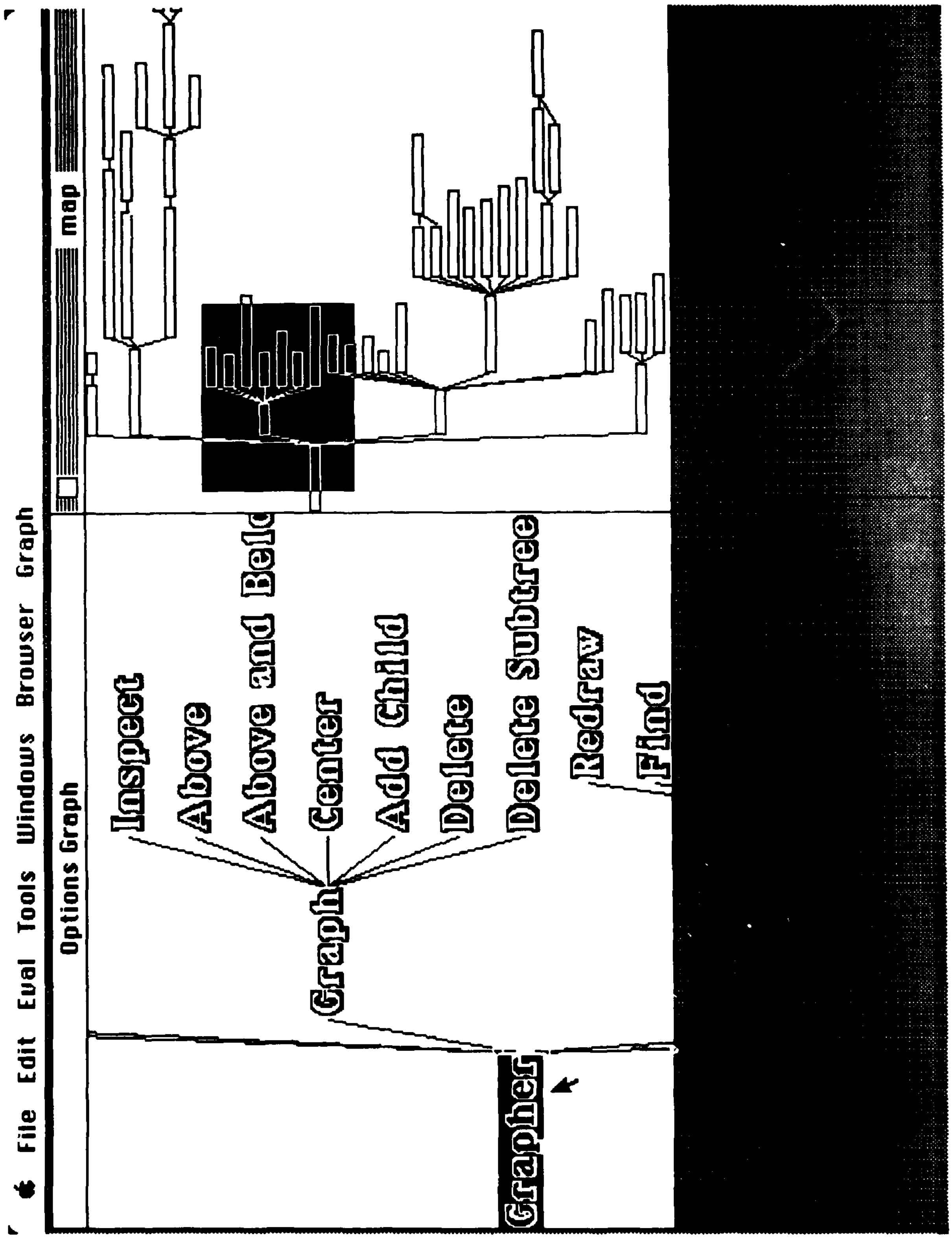


Figure 19: Here yet another font change is illustrated. The graph depicted here is actually some of the high-level functionality of the ISI Grapher itself. This is an example of using the ISI Grapher to graph function-calling hierarchies.

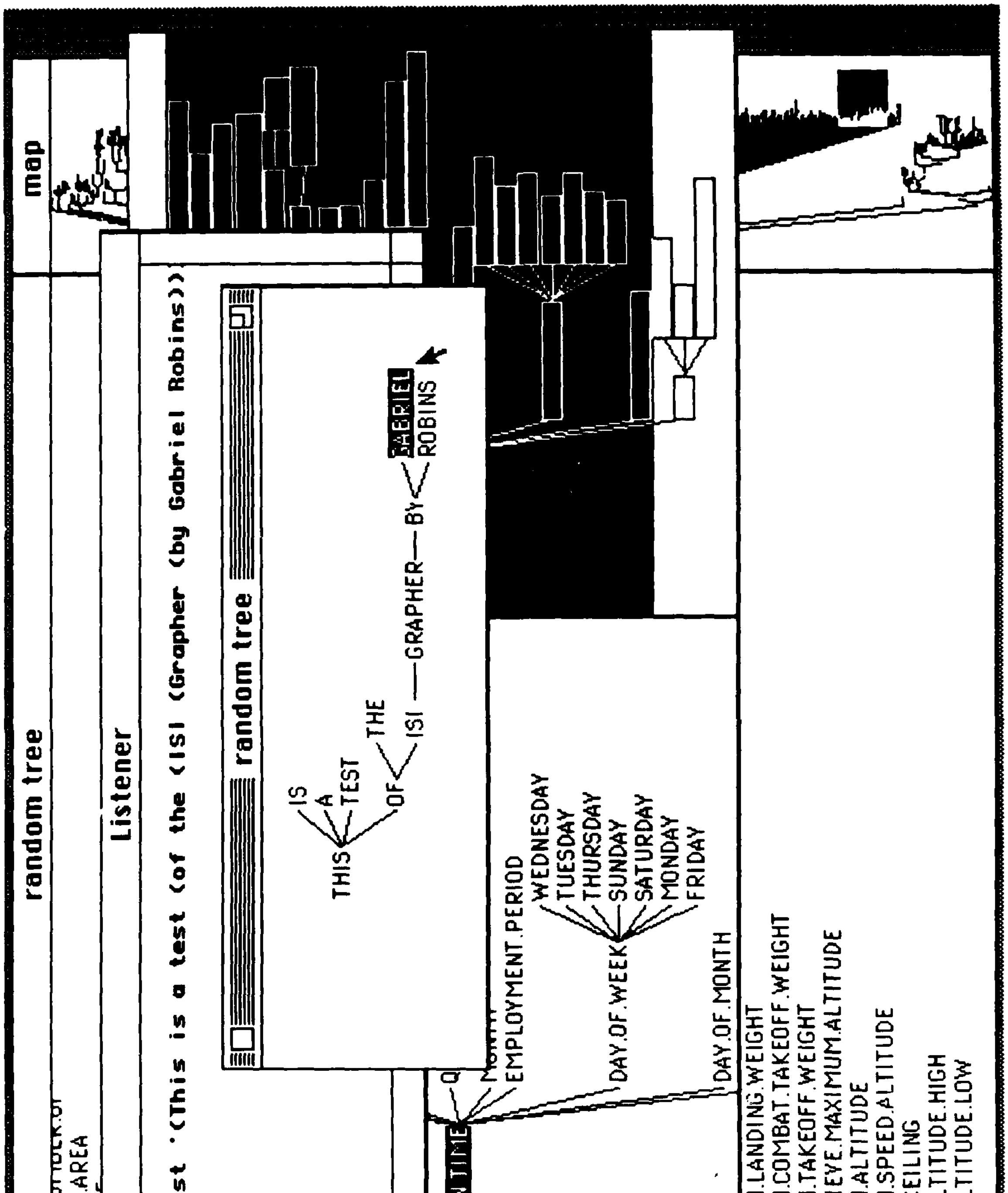


Figure 20: Here the List Grapher was invoked in a LISP Listener; the small sample list which was entered has been graphed in a separate window. The List Grapher is a simple tool built on top of the ISI Grapher, and is used to graph arbitrary lists (the CAR being the root, and the CDR being the list of children, recursively).

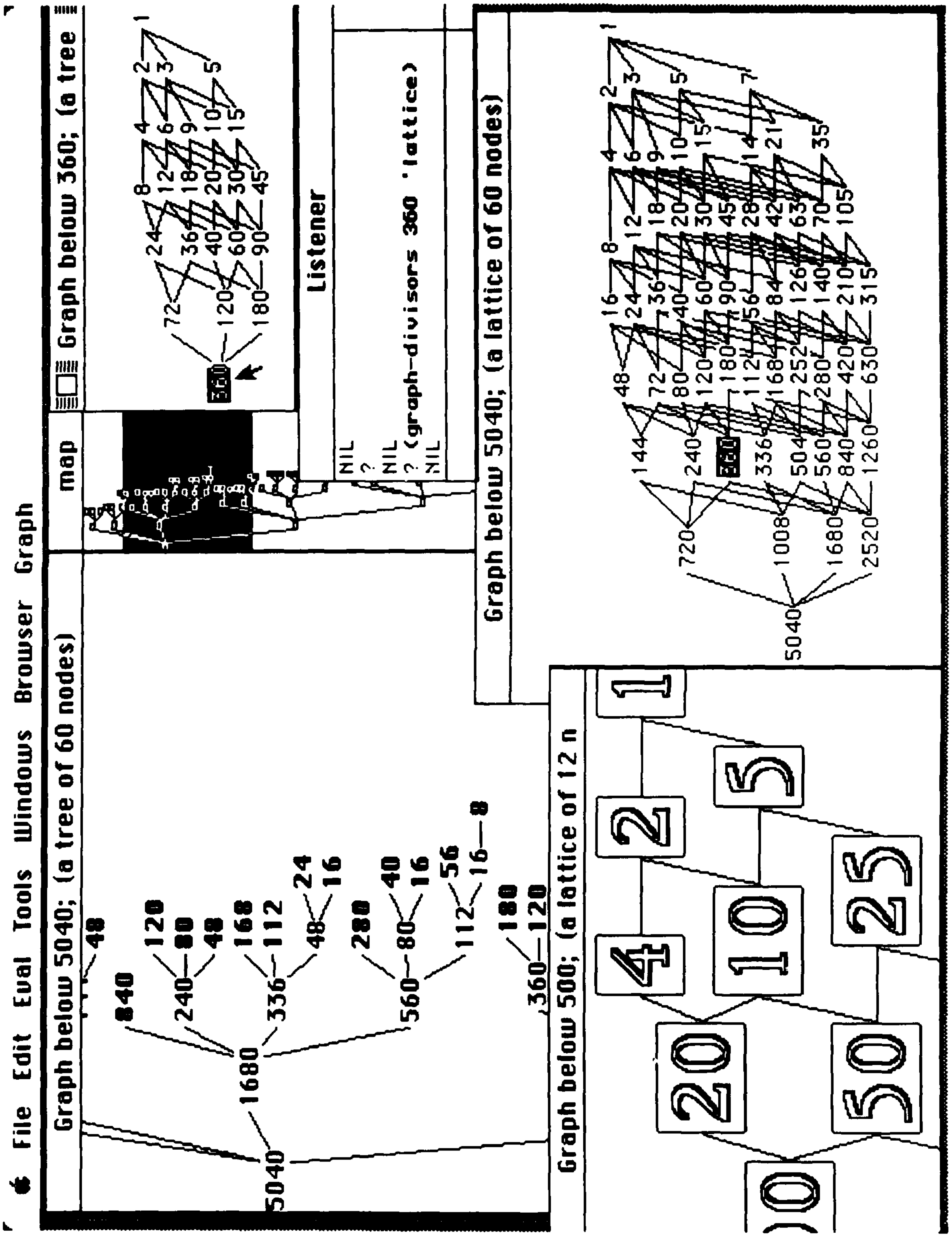


Figure 21: Here the Divisors Grapher is illustrated; given an integer, it displays the divisibility partial order for that integer (nodes represent divisors and edges represent the relation "is divisible by").

21. Index

FILMED

6 8